UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

IWVI

Mapping Graph- and Logic-based Process Model Query Language Features

Masterarbeit

Zur Erlangung des Grades eines Master of Science im Studiengang Wirtschaftsinformatik

vorgelegt von

Sebastian Schlicht

[211 100 329]

Koblenz, im September 2019

Erstgutachter:	Prof. Dr. Patrick Delfmann
	(Institut für Wirtschafts- und Verwaltungsinformatik, FG Delfmann)
Zweitgutachter:	M.Sc. Carl Corea
	(Institut für Wirtschafts- und Verwaltungsinformatik, FG Delfmann)

Eidesstattliche Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen - benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

(Ort, Datum) (Unterschrift)

Zusammenfassung

Business Process Querying (BPQ) ist eine Disziplin im Bereich des Geschäftsprozessmanagement, die Experten beim Verständnis vorhandener und der Entwicklung neuer Geschäftsprozesse unterstützt. Anfragen können Modelle identifizieren und vereinigen, Fragen bezüglich zugrunde liegender Geschäftsprozesse beantworten und Prozesse daher auf Konformität prüfen. Viele Anfragesprachen wurden im Rahmen dieser Disziplin eingesetzt doch zwei Sprachtypen dominieren: Logik-basierte Sprachen nutzen temporale Logik, um die Modelle in Form endlicher Zustandsautomaten zu verifizieren. Graph-basierte Sprachen nutzen wiederum musterbasierte Suchen, um passende Teilgraphen direkt aus dem Modellgraph zu extrahieren. Diese Arbeit zielt darauf ab, die Funktionen beider Spracharten aufeinander abzubilden, um ihre Vor- und Nachteile zu identifizieren. Hierfür werden die Funktionen von Computational Tree Logic (CTL) und The Diagramed Modeling Language (DMQL) exemplarisch aufeinander abgebildet. CTL berechnet zuerst alle validen Zustände und ist daher für behavioristische Anfragen besonders geeignet. Mangels bestimmter struktureller Funktionen und Zählmechanismen ist CTL jedoch kaum für strukturelle Anfragen anwendbar. Im Gegensatz dazu führt DMQL strukturelle Abfragen aus, deren Muster jede CTL-Formel darstellen können. Diese erzielen jedoch nicht zwingend exakt dieselbe Semantik: Muster behandeln bedingten Programmfluss stets wie sequenziellen, da sämtliche Bedingungen ignoriert werden. Im Ergebnis stellen die extrahierten Pfade in bestimmten Szenarien ungültigen Programmfluss dar und sind daher Fehlalarme. DMQL kann für behavioristische Anfragen genutzt werden wenn diese nicht vorhanden bzw. die Fehlalarme akzeptabel sind. Zusammenfassend haben beide Spracharten Stärken und sind auf verschiedenen Anwendungsfälle in BPQ spezialisiert doch graph-basierte Sprachen können ggf. für beide genutzt werden. Wird die Evaluation von Bedingungen noch integriert, könnte die Notwendigkeit für logik-basierte Sprachen in BPQ künftig entfallen.

Abstract

Business Process Querying (BPQ) is a discipline in the field of Business Process Management which helps experts to understand existing process models and accelerates the development of new ones. Its queries can fetch and merge these models, answer questions regarding the underlying process, and conduct compliance checking in return. Many languages have been deployed in this discipline but two language types are dominant: Logic-based languages use temporal logic to verify models as finite state machines whereas graph-based languages use pattern matching to retrieve subgraphs of model graphs directly. This thesis aims to map the features of both language types to features of the other to identify strengths and weaknesses. Exemplarily, the features of Computational Tree Logic (CTL) and The Diagramed Modeling Language (DMQL) are mapped to one another. CTL explores the valid state space and thus is better for behavioral querying. Lacking certain structural features and counting mechanisms it is not appropriate to query structural properties. In contrast, DMQL issues structural queries and its patterns can reconstruct any CTL formula. However, they do not always achieve exactly the same semantic: Patterns treat conditional flow as sequential flow by ignoring its conditions. As a result, retrieved mappings are invalid process execution sequences, i.e. false positives, in certain scenarios. DMQL can be used for behavioral querying if these are absent or acceptable. In conclusion, both language types have strengths and are specialized for different BPQ use cases but in certain scenarios graph-based languages can be applied to both. Integrating the evaluation of conditions would remove the need for logic-based languages in BPQ completely.

Contents

List of Figures

List of Tables

Acronyms

BP	Business Process
BPM	Business Process Management
BPMN	Business Process Model and Notation
BPQ	Business Process Querying
CC	Compliance Checking
CTL	Computation Tree Logic
DMQL	Diagramed Model Query Language
GMQL	Generic Model Query Language
LTL	Linear Temporal Logic
PLTL	Past Linear-time Temporal Logic
VMQL	Visual Model Query Language

1. Introduction

Business Process Management is based on the explicit representation of business processes, i.e. creating models of a company's real-world processes. As a result, BPs are represented digitally and can be further processed by information systems. Business Process Querying (BPQ) is a subfield of Business Process Management where structural, behavioral and semantic properties of business processes are checked for compliance with given queries. Querying business process models can speed up the design of new process models, help to understand or merge existing models and answer questions regarding a company's processes. The latter is interesting for customers (e.g. for choosing future business partners) and the company itself, as it allows for automated compliance checking. Automated compliance checking of business processes is important, facing the vast number of ever-changing legal constraints, industrial standards and internal policies that a process might need to fulfill. For example, mobile operators might be legally required to verify the identity of customers before providing and activating the SIM card. In BPQ this translates to checking the absence of an undesired BP element sequence, i.e. where an activation is not proceeded by an identity verification.

Clearly, the power of BPQ is determined by the employed query language. Various languages have been developed and deployed for BPQ. Two types of languages dominate the field: Languages based on temporal logic have been primarily developed for system verification and transform processes into finite-state machines to evaluate them against defined formulas. Languages using graph pattern matching retrieve subgraphs by matching given query patterns to process model graphs. We refer to these languages as logic-based and graph-based languages, respectively. There is an ongoing discussion of the advantages and disadvantages concerning the two language types, especially when querying process models, i.e. the definitions of processes: Some authors believe that structural querying using pattern matching can only approximate behavioral querying on an instance level, i.e. system verification (c.f. (Beeri et al. 2008)). Others state that behavioral semantics can be expressed with temporal logic and graphs (c.f. (Wang et al. 2014)) which both consequently must allow behavioral querying.

This thesis aims to investigate whether logic- and graph-based languages provide distinct features for querying process definitions or rather can be expressed by one another. Concretely, the following questions have driven the present research:

RQ1 What are the advantages of logic-based compared to graph-based business process query languages and vice-versa?

- RQ1a Which features do these languages offer?
- RQ1b Which features of either language type can be emulated by features of the other?

RQ1c Which features are left unique to either language type?

To answer these research questions, the features of a popular temporal logic (Computation Tree Logic (Clarke et al. 1986)) and a generic, graph-based query language (The Diagramed Model Query Language (Delfmann, Breuker, Matzner & Becker 2015)) are described exemplary, summarized and mapped to each other where possible. Finding answers to our questions contributes to this discussion and may guide the development of future query languages to close the feature gap and potentially end the discussion. The findings of this work are that pattern matching can reconstruct any formula of (branching-time) temporal logic but retrieves false positives and false negatives (i.e. invalid executions) in certain scenarios. These validity issues can be resolved manually and may conceptually be resolved in future work. Temporal logic, on the other hand, largely fails to reconstruct the feature pattern matching offers.

Chapter 2 introduces the problem field, delimits the problem context from related concepts and provides the necessary background for temporal logic and graph pattern matching. Chapter 3 presents the mapping from temporal logic to graph querying language features and vice versa. Chapter 4 summarizes and discusses the findings of the previous chapter. Finally, chapter 5 concludes on this thesis, points at its limitations and future work, e.g. regarding the validity issues.

2. Background

This section presents the background required to assess and reason about the defined research questions. Firstly, the problem description is narrowed down, starting with the parental problem field. Section 2.1 introduces the problem field Business Process Management and the problem context Business Process Querying, being a part of this field. Besides introducing the required concepts the section delimits the problem context from related concepts within the problem field. Next, a rudimentary foundation of the techniques which are commonly applied to the two types of query languages is given. Logic-based querying deploys temporal logic (section 2.2) while graph-based querying instruments graph pattern matching (section 2.3). In particular, both sections list the application of available languages within the problem context. For each query language type we select one representative language. The selected languages are used in chapter 3 to map the features of one type of query languages to the other.

2.1. Business Process Management (BPM)

Business Process Management (BPM) includes concrete techniques up to higher level concepts supporting all phases in the lifecycle of a business process (BP), including its design, administration, enactment and analysis (Weske 2012). The goals of BPM include (1) to provide a better understanding of the company operations and their relationships, (2) to increase the speed of changes and (3) to capture knowledge about the business in form of a BP repository. However, BPM is not limited to these goals and actually solves additional problems, Weske provides a comprehensive overview. At the very heart of these definitions are BPs. BPs are collections of activities, performed in coordination to jointly realize the business goal (Weske 2012). In this context the BP is an instrument to explicitly represent the respective real-world process (or alternative versions), as each product (or service) is an outcome of a series of activities (Hammer & Champy 1993). This explicit representation of BPs is fundamental to BPM and how it achieves its goals. The representation itself is referred to as BP definition or model and contains the activities of the BP together with their execution constraints (Weske 2012). As such, it is a blueprint for business cases and is used to implement the BP manually or using software systems (Weske 2012). Organizational rules and policies implement a process manually but software can actually enforce execution constraints within its activities (Weske 2012). While BPs are enacted by a single organization, they are performed within an organizational and technological environment where they possibly interact with other BPs (process choreography) (Weske 2012). A Business Process Management System (BPMS) is a piece of a software which coordinates the enactment of BPs, driven by their explicit representation. While activities of enacted BPs can be executed manually, e.g. by knowledge workers using graphical interfaces, enactment often includes the automation of activities. Thus, BPMSs also integrate involved IT systems and document this integration within the BP.

Section 2.1.1 describes a commonly used visual notation for the explicit representation of BPs. Finally section 2.1.2 summarizes how explicit representations of BPs (e.g. expressed in such notations) can be queried and how this querying contributes to the goals of BPM.

2.1.1. Business Process Model and Notation (BPMN)

The Business Process Model and Notation (BPMN) is a metamodel of and notation for business processes (currently in version 2) (Group 2014). BPMN has been developed by the Object Management Group[1] (OMG), an open not-for-profit consortium for technology standards driven, among others, by vendors (e.g. SAP, IDS Scheer, Software AG), end-users and academic institutions (Group 2019). BPMN defines several diagram types for BPs and is organized in different layers, where the notation allows to visualize elements from the respective metamodel. To present the full BPMN standard is out of scope of this work. Instead, this section focuses on basic modeling elements of Business Process Diagrams which features four categories of elements (Weske 2012):

1. flow objects
2. artefacts
3. connecting objects
4. swimlanes

We briefly introduce each category and continue with introducing BPQ for BP models, e.g. modeled in BPMN. An example BPMN model is presented and discussed in chapter 4.

2.1.1.1. Flow Objects

Flow objects are embedded in the control flow of a BP. The control flow describes the theoretically possible execution of each BP case, from start to end. There are three types of flow objects: (1) events, (2) activities and (3) gateways.

Events Events represent states and incidents, both physical and electronic, that can be mapped to the real world. An example are incoming messages. They are visualized using circles with different types of borders and icons, as shown in figure 2.1. The border indicates the position type of an event. Events can be at the start, at the end or in between the start and the end of a BP. The latter position type of events is called intermediate events. Though it is best practice to start and end BPs with dedicated start and end events, it is not mandatory in BPMN (Weske 2012). The icon inside the event shape indicates the (optional) event type. For example, *message* events mainly indicate the receiving of

[1]https://www.omg.org

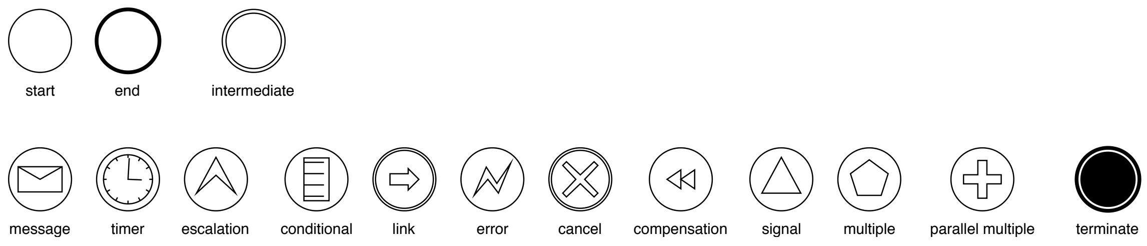


Figure 2.1.: BPMN event shapes in the interrupting flavor.

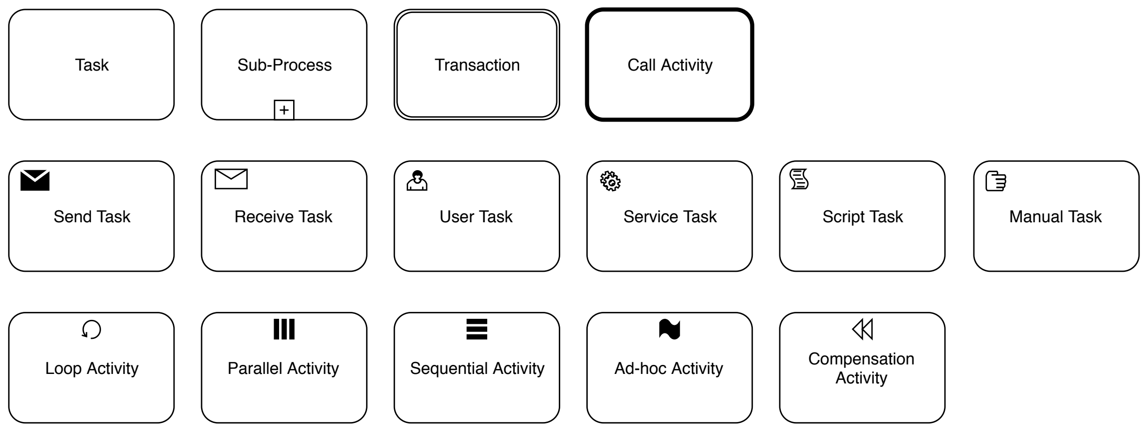


Figure 2.2.: BPMN activity shapes.

messages and *timer* events trigger after certain temporal conditions are met (e.g. hourly or after 12 hours passed). However, there are numerous event types and not all types exist for every position type (c.f. (Weske 2012)). Additionally, events can have different flavors. Events usually are interrupting, i.e. they listen and wait until the respective incident occurs, such as the start event. Events can also be non-interrupting (event shapes with dashed borders) or throwing (event shapes with filled icons). Latter events are actively triggered by the process during its execution, such as the end event.

Activities Activities specify units of work which may or may not be further divided (Weske 2012). They are visualized as rounded rectangles, as shown in figure 2.2. Atomic activities are referred to as tasks. These are reasonably small units of work that are to be executed by a BP participant or an involved system. Task rectangles feature a single border and may show an icon in their upper left corner. This icon indicates the optional type of the task. For example, tasks can send or receive information, user tasks involve human interaction and service tasks are fully automated by IT systems instead. Manual tasks allow to model tasks which can not be supported by any involved system at all. In contrast, user tasks involve human interaction but are supported by systems with graphical interfaces and usually generate data to be stored. However, activities can also be non-atomic and thus not only tasks. A plus icon in the rounded rectangle indicates a subprocess activity which is described by a separate BP and its activities. There are also transaction activities and call activities which require additional, possibly remote actions to complete before the activity can be marked as completed. They are visualized as activities with double and bold borders, respectively. In addition, there is a number of markers activities can be annotated with. The loop marker expresses that the activity is iterated depending on

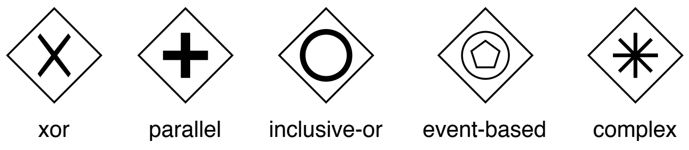


Figure 2.3.: BPMN gateway shapes.

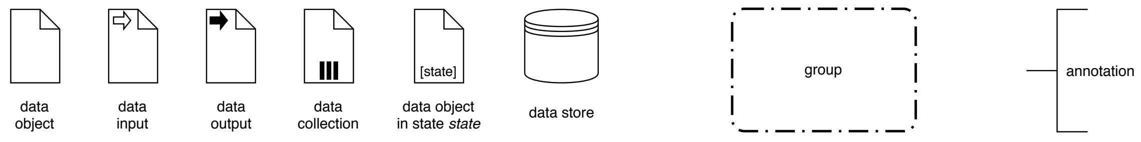


Figure 2.4.: BPMN artefact shapes.

certain attributes. Other markers mark activities as sequential or parallel, ad-hoc activities that are not embedded in the control flow or compensation activities which undo specific prior activities.

Gateways Gateways define points in a BP model where control flow branches or joins. Thus conditional control flow usually is modeled via gateways. They are visualized by diamonds where the icon inside the diamond indicates the gateway type, as shown in figure 2.3. BPMN does not separate between branching and joining gateways but this property can be derived from the number of outgoing flows. There are five types of gateways: Exclusive-or gateways (x symbol) activate only a single of its outgoing branches. Parallel gateways (plus symbol) activate all branches and synchronize them when joining. Inclusive-or gateways (circle) activate any number of outgoing branches instead. The previous gateways decide which branches to activate themselves. Event-based gateways (pentagram) are similar to *xor* gateways but let the execution environment decide which branch to activate. Complex gateways (asterisk symbol) allow to define the split and join behavior via activation conditions and gateway expressions. Though joining gateways are not necessarily required in BPMN, it is good practice to use gateways for both branching and joining.

2.1.1.2. Artefacts

Artefacts represent resources which add extra information to the model that is not related to the control flow. Supported artefacts are (i) data objects, (ii) groups and (iii) annotations. The different shapes are presented in figure 2.4. Artefacts may be arbitrarily connected to flow objects.

Data Objects Data objects represent information or physical objects such as invoices or parcels. In BPMN data objects document the usage of data in a process and can not define the internal structure of the actual data (Weske 2012). Primarily, they are represented by rectangles with an earmark. Arrow icons in the upper left corner may indicate whether the data object is used as an input (such as customer orders) or output (such as an invoice) of a flow object. A subscript icon with three vertical bars marks data objects which can actually be broken down into multiple objects, as such the object is a collection of objects. The state of a data object can be expressed within square brackets inside the shape. Finally,

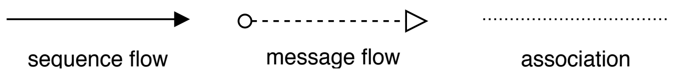


Figure 2.5.: BPMN connection object shapes.

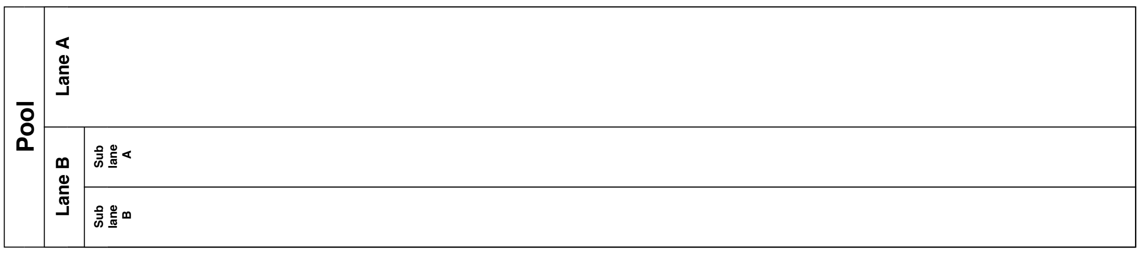


Figure 2.6.: BPMN pool and swimlane shapes.

the data store provides means to read and write data to a persistent storage (c.f. (Cruz et al. 2012)).

Groups and Annotations The last types of artefacts are groups and annotations. Groups can be used to group elements for the sake of layouting and without additional meaning. Thus, they are allowed to span any parts of a model. Annotations allow to place textual documentation inside the model, e.g. to further explain a certain object.

2.1.1.3. Connection Objects

Connection objects represent relationships between flow objects and/or artefacts. There are different types of connections, each having their own arrow style as visualized in figure 2.5. Sequential flow specifies the ordering of flow objects, such as events and activities, and is represented by solid arrows. Sequential flow may be conditional, if it is flow is constrained by the values of certain attributes. Message flow is displayed as dashed arrows and connects flow objects with artefacts, such as data objects. It indicates the flow of messages from sender to receiver. Finally, *associations* related artefacts to other elements, indicated by a dotted line.

2.1.1.4. Swimlanes

Finally, swimlanes structure a model according to its organizational environment. Swimlanes are organized within a pool having any number of lanes where each lane represents a particular organization. Swimlanes are displayed as rectangles and solid lines divide them into the different lanes, as shown in figure 2.6. Sublanes allow the model to visualize more fine-granular organizational hierarchies.

2.1.2. Business Process Querying (BPQ)

Business Process Querying (BPQ) is a discipline in the field of BPM studying methods to query BPs in a repository (c.f. (Polyvyanyy 2018), (Polyvyanyy et al. 2017)). Querying, in this context, is the execution of a process query, a formal instruction which can read and/or write (parts of) BPs inside the repository. However, this work is limited to read queries and we use the term BPQ to refer to this subset from now on. More specifically, we restrict BPQ to the analysis of BP models, for example modeled in BPMN. BPQ helps to understand BP models, find and extract (frequent) patterns for reusing and check a BP for certain properties (Awad 2007). Thus, BPQ is a subfield and technique that supports to reach the goals of BPM. Extracting frequent patterns reduces redundancy. Extracting patterns in order to reuse model parts is even more important: Model development is complex, time consuming and error prone (Awad, Decker & Weske 2008, Wang et al. 2014). Instead of starting from scratch, reusing existing model parts can save time and costs. BPQ is required for the understanding and extraction of these parts, due to large model sizes and the rapid growth of the number of models per company, continuously making manual methods more impractical (Awad, Decker & Weske 2008). Especially when BPs have to be merged, as in case of company amalgamations (Wang et al. 2014). Furthermore, a BP captures important information about a business and how it is conducted (Wang et al. 2014). This is why it can be used as an information mine, e.g. to answer (customer) questions regarding the underlying business process (c.f. (Beeri et al. 2008)). More prominently real-world constraints, such as legal regulations, are translated into properties that a BP can be checked for. Compliance checking (CC) is the name of this third major use case for BPQ. Section 2.1.2.1 puts BPQ in the context of BPM and delimits it from related concepts such as compliance checking. The different goals of BPQ yield different querying types which are presented in section 2.1.2.2. Both led to the development of numerous query languages. Section 2.1.2.3 provides a brief overview about query languages used for BPQ.

2.1.2.1. Delimitation of Related Concepts

BPQ is a discipline in the field of BPM and thus is related to many concepts in this field. For example, BPQ can be seen as an approach to Business Process Analysis (BPA) and an enabler of Business Intelligence (BI) (Riehle 2018). To give a contextual overview about BPQ and to delimit BPQ from related concepts, they will be briefly introduced in this section.

Compliance checking (CC) is the field of checking BP models for compliance with certain rules. These rules describe regulations, policies and quality constraints the BP must meet (Awad, Decker & Weske 2008). Typically, they ask for certain ordering or presence/absence relations between activities. BPQ is required for automated CC as the rules are frequently changing and manual checking is time-intensive (Awad, Decker & Weske 2008). CC is separated into forward and backward CC (El Kharbili et al. 2008). Forward CC is proactive and aims to verify the rules at design-time or post design to prevent non-compliant behavior. Backward CC is reactive and aims to detect non-compliant behavior by analyzing the execution history. BPQ is primarily used in forward CC (post design).

Process Mining (PM) is a research field with the aim to discover structures from the execution history of BPs (c.f. (van der Aalst et al. 2005)). The motivation of PM stems from the property that BPs leave footprints in all process-related systems (i.e. their logs) (van der Aalst et al. 2005). Thus it provides techniques and tools to analyze these logs and discover structures such as processes (process discovery), i.e. it learns the definition of a BP based on the execution history of relevant cases. However, PM also includes (backward) CC and bottleneck analysis, among others (Polyvyanyy et al. 2017). PM is closely related to BPA and aims to bride the gap between data-driven BI and the process-centric BPM (Polyvyanyy et al. 2017). In theory BPQ can be applied in the context of PM, e.g. when querying logs or models derived via process discovery.

BPQ is generally defined as the querying of BPs. Confusingly, some authors speak of workflows in this context (c.f. (Smith et al. 2012)). Workflows refer to the automation of a BP (in part or at a whole) according to procedural rules (Weske 2012). Workflow Management is the respective field and involves an explicit representation of BPs, as in BPM, plus their controlled enactment based on the given models (Weske 2012). As such, workflows are enacted BPs and in result these terms often are interchangeable.

2.1.2.2. Querying Types

There are different approaches to categorize the field of BPQ. One approach is to categorize methods by the concrete BP entity that is being queried, which depends on the stage of the BP (c.f. (Awad 2007, Polyvyanyy 2018)). At first, there is the definition of the BP, e.g. captured in a model. Querying the definition involves to search for patterns in the control flow or to lookup data requirements. The aim of this BPQ type primarily is to get a better understanding of the BP or to reuse existing processes before designing new ones. Secondly, the definition is instantiated yielding running BP instances. Querying running instances allows monitoring the cases and query their status, e.g. to detect deadlocks. Thirdly, the BP completes and leaves an execution history (i.e. a log). Querying the history can be used for process discovery but also aims to generate statistics about the process duration and bottlenecks (Awad 2007). Usually this querying type is used when a definition is lacking (Polyvyanyy 2018).

Another approach for structuring BPQ methods is to analyze the type of information they are processing, referred to as the perspective (c.f. (Wang et al. 2014, Polyvyanyy et al. 2017)). Structural querying focuses the structural topology of the model (i.e. the definition). Behavioral querying focuses on behaviors induced by the activity models, which refers to the implicit execution semantics of behavioral models. At the first glance both categorization approaches may look equal. Querying the BP definition primarily involves structural querying (Wang et al. 2014) while querying running instances is a behavioral analysis. However, the querying stages do not imply the applied perspective and both are separate dimensions, indeed. This is because the BP definition carries implied behavioral semantics (in its notion) and thus allows for behavioral querying, to some extent (Wang et al. 2014). In fact, querying the definition approximates behavioral querying and can at least narrow the search space for the costly verification behavioral querying implies (Beeri et al. 2008).

Additionally, there are different types of querying which span all stages and perspectives. For example, semantic querying refers to the usage of similarities, an ontology etc. to abstract from exact matches (in terms of activity labels or structures) and focus on underlying meanings instead. However, this work focuses on the structural and behavioral querying of BP models (i.e. definitions).

2.1.2.3. Languages

The different goals and perspectives of BPQ led to the development of different querying techniques and numerous query languages for BP models. An overview and categorization of available languages is given in (Polyvyanyy et al. 2017). Dominant techniques are pattern matching (graph-based) and model checking (Clarke Jr et al. 2000) techniques using temporal logic (logic-based) (Riehle 2018), especially in the field of structural and behavioral BP definition querying. BP-QL (Beeri et al. 2006), DMQL (Delfmann, Steinhorst, Dietrich & Becker 2015), GMQL (Delfmann, Breuker, Matzner & Becker 2015) and VMQL (Storrle 2009) are examples for graph-based query languages and will be presented in section 2.3.1. Languages of this category are primarily used for structural querying, as they query the BP definition directly (Riehle 2018). In contrast, logic-based languages are also more commonly used for behavioral querying, including the verification of compliance rules. Logic-based languages incorporate temporal logic such as LTL (Pnueli 1977), PLTL (Zuck 1986), CTL (Clarke & Emerson 1981) or MTL (c.f. (Polyvyanyy 2018)). For example, BPMN-Q (Awad 2007) is a visual query language based on (P)LTL. In addition to these dominant categories there are declarative query languages such as PQL (Kammerer et al. 2015) as well as semantic languages such as APQL (Ter Hofstede et al. 2013), both used for behavioral analysis (Polyvyanyy 2018). A more comprehensive list of available languages is provided by dedicated surveys, such as (Polyvyanyy et al. 2017).

2.2. Temporal Logic

Temporal logic provides a formalism for describing assertions about event occurrences in time - possibly past, presence and future - to reason about concurrent programs (Emerson & Halpern 1986). Starting in the last century, it has been used as an unified approach for the verification of programs and systems (Pnueli 1977). There are two views on temporal logic which differ in their understanding of the nature of time and are used to classify systems into categories (Emerson & Halpern 1986). In linear temporal logic, time is linear and there is only one possible future. In branching time logic, time may branch into alternative futures like a tree. The chosen view is a boundary for the expressiveness of related systems and determines the set of allowed temporal operators and modifiers. There are several common temporal languages, each covering different parts of the linear, the branching or even both views on time. Linear Time Logic (LTL) was proposed for formal verification in (Pnueli 1977) and refers to linear time temporal logic with solely linear time operators. Computation Tree Logic (CTL) is a branching time temporal logic where path quantifiers must prefix linear time assertions (Clarke & Emerson 1981, Browne et al. 1988). CTL* is an extension of CTL where path quantifiers can prefix linear time assertions (Emerson & Halpern 1986). Thus, CTL* combines linear and branching time logic to a powerful, more expressive temporal logic (Browne et al. 1988). Past linear time temporal logic (PLTL) has been introduced in (Zuck 1986) and is a LTL with counterpart operators to refer to past states (Awad, Decker & Weske 2008).

In temporal logic, a formula is evaluated against a typically finite system (i.e. having a finite number of states). Model checking is a common approach to automatically verify these finite state systems against temporal logic formulas (or specifications) (Clarke Jr et al. 2000). The system is modeled as a state-transition graph which is then analyzed and verified against the formula. Model checking, as initially proposed, suffers from the state explosion problem and is not sufficient for large systems (i.e. having a large number of states) (Clarke Jr et al. 2000). Symbolic model checking uses dedicated data structures (e.g. binary decision diagrams) to cope with the state explosion problem (Clarke Jr et al. 2000). These structures e.g. abstract from data values by mapping actual values to sets of abstract values. This abstraction is then extended to states and state transitions to generate and verify the resulting abstract system, often having a much smaller size. Model checking, however, typically is not restricted to the initial variant but refers to any technique alike, including symbolic model checking.

Section 2.2.1 provides an overview about the usage of temporal logic in BPQ and finishes with a brief discussion on their appropriateness. An overview of applied temporal logic languages has already been presented in (Elgammal et al. 2016). CTL is one of these languages and has been selected as the most appropriate temporal logic for BPQ. Hence CTL will be used as the reference language for logic-based BPQ. Section 2.2.2 introduces CTL with its operators and their semantics.

2.2.1. Logic-based BPQ

When temporal logic is used for BPQ, the respective query is transformed into a temporal logic expression (i.e. formula), defining the desired properties (Polyvyanyy 2018). In order to analyze the BP with temporal logic, a finite state system is required. Depending on the used modeling language this calls for pre-processing and transformations (Riehle 2018). In principle, the BP model is transformed into a finite state system and evaluated against the formula and its implied properties using model checking (De Nicola et al. 2012, Polyvyanyy 2018). However, this transformation may include intermediate transformations into more formal modeling languages such as Petri nets (Reisig & Rozenberg 1998), depending on the exact modeling language (De Nicola et al. 2012).

There are multiple examples for BPQ based on temporal logic, mainly in the area of CC. BPMN-Q is a full query language for BPMN models with a BPMN-like syntax (Awad 2007). Though BPMN-Q is primarily a visual language for structural querying, it incorporates PLTL model checking to support (behavior-related) CC (Awad, Decker & Weske 2008).

BPMN-Q has also been combined with CTL to conduct compliance checking (Awad et al. 2011). Visual BPMN-Q queries were used to express compliance requirements which were mapped to CTL in a subsequent step, to take behavioral aspects into account. Additionally, BPMN-Q has been expanded to support automated semantic querying using generated ontologies (Awad, Polyvyanyy & Weske 2008). LTL has been used to verify properties of executed BP based on log entries in (van der Aalst et al. 2005). (Liu et al. 2007) uses model checking via LTL to check BPEL models for compliance with constraints originally defined in the Business Property Specification Language (BPSL). Förster et al. used PLTL for compliance checking on UML Activity Diagrams in a series of papers (c.f. (Förster et al. 2007)). The constraints have been originally defined using the visual Process Pattern Specification Language (PPSL). In (Elgammal et al. 2016) a Compliance Request Language (CRL) is specified grounded in LTL, to express common CTL and LTL expressions used in CC more compactly (Riehle 2018). Still closely related to BPQ is the usage of temporal logics in declarative modeling languages. For example, DECLARE (Pesic & Van der Aalst 2006, Pesic et al. 2007) allows the declarative modeling of BPs where visual elements represent constraints and (invisibly to the modeler) map to temporal logic constructs.

Facing this set of related work raises the question which temporal logic to use for BPQ. At first, there are formulas in LTL which can not be expressed using CTL and vice versa (Vardi 2001). Arguably, LTL is the preferred choice when querying linear entities such as logs of executed BPs where there is a linear sequence of events (van der Aalst et al. 2005). However, prior to the execution of BPs linearity is not given and multiple branches may be enabled in the future. In contrast to LTL, CTL naturally represents and works with these branches. This is why CTL provides a natural means to express acceptable behaviors (Venkatraman & Singh 1999). Finally, model checking seems to be less complex for branching than for linear time logic (Emerson & Halpern 1986). Thus, CTL is selected as the reference language for logic-based BPQ.

2.2.2. Computation Tree Logic (CTL)

Computation Tree Logic (CTL) (Clarke & Emerson 1981) is a subset of CTL* where only branching-time operators are permitted, i.e. linear-time operators must be prefixed by path quantifiers (Browne et al. 1988). This is because, unlike in many other temporal logic languages, CTL respects the branching nature of time and is aware of multiple possible futures. Formulas need to express whether a linear-time formula should be valid on one or all these futures. As in every temporal logic, you construct formulas that may be evaluated on finite state systems and, in result, be used for model checking (Clarke & Emerson 1981). The syntax of CTL and its formalization is presented in section 2.2.2.1, based on Kripke structures. Section 2.2.2.2 describes the semantics of the CTL operators and related formulas in greater detail.

2.2.2.1. Syntax and Formal Semantics

Given a system consisting of states, CTL internally allows to express two types of formulas, namely (1) state formulas that are evaluated in a specific state and (2) path formulas that are evaluated along paths, infinite sequences of states. Though both types exist, only state formulas are valid CTL formulas. To become state formulas, path formulas must be prefixed with path quantifiers (see above). In order to introduce the syntax of the state formulas in CTL, it is required to understand that each state is associated with atomic propositions that either do or do not hold in the respective state. Given a set of atomic proposition names AP, a state formula is either

1. $a \in AP$ - the name of an atomic proposition;
2. $\neg f_1, f_1 \vee f_2$ - a logical transformation or combination of state formula(s) f_1, f_2;
3. $E(g)$ - a path formula g prefixed with a path quantifier.

In this context, a path formula is either

1. $X f_1$, $f_1 U f_2$ - a branching time operator on state formula(s) f_1, f_2;
2. $\neg g$ - the negation of a path formula g.

The semantics of CTL are defined with respect to a Kripke structure $M = (S, R, L)$ and the set of atomic proposition names AP (Clarke & Emerson 1981, Clarke et al. 1986) where

1. S is a set of states,
2. $R \subseteq S \times S$ is the transition relation and
3. $L : S \rightarrow 2^{AP}$ is a labeling function that determines the set of valid atomic propositions $L(s)$ in state $s \in S$.

For a more compact description of the semantics, additional definitions have proven useful. We use $s_1 \rightarrow s_2$ to denote that $(s_1, s_2) \in R$. Let $\pi = s_0, s_1, ...$ be a path, i.e. an infinite sequence of states, in M where $\forall i \in N_0 : s_i \in S, s_i \rightarrow s_{i+1}$. In this context, π^i denotes the suffix of path π starting at s_i and π_i denotes the i-th elements of π, i.e., s_i. The model relation $\models$ is now used to indicate that a state or path formula holds at a state or along a path and inductively defined, given state formulas f_1, f_2 and path formulas g_1, g_2:

(R1) $s \models a \quad \Leftrightarrow \quad a \in L(s)$

(R2) $s \models \neg f_1 \quad \Leftrightarrow \quad s \not\models f_1$

(R3) $s \models f_1 \vee f_2 \quad \Leftrightarrow \quad (s \models f_1) \vee (s \models f_2)$

(R4) $s \models E(g_1) \quad \Leftrightarrow \quad \exists \pi : (\pi_0 = s \wedge \pi \models g_1)$

(R5) $\pi \models f_1 \quad \Leftrightarrow \quad \pi : (\pi_0 = s \wedge s \models f_1)$

(R6) $\pi \models \neg g_1 \quad \Leftrightarrow \quad \pi \not\models g_1$

(R7) $\pi \models g_1 \vee g_2 \quad \Leftrightarrow \quad (\pi \models g_1) \vee (\pi \models g_2)$

(R8) $\pi \models X(g_1) \quad \Leftrightarrow \quad \pi^1 \models g_1$

(R9) $\pi \models g_1 U g_2 \quad \Leftrightarrow \quad \exists k \in N_0 : \big(\pi^k \models g_2 \wedge \forall (j \in N_0 | j < k)(\pi^j \models g_1)\big)$

This inductive definition is a direct translation of the set of rules presented above. While the definition is complete, i.e. represents a minimal subset that any CTL formula can be expressed with, there are additional logical symbols and time operators which are commonly defined based on existing ones:

(A1) $A(g_1) \quad \Leftrightarrow \quad \neg E(\neg g_1)$

(A2) $f_1 \wedge f_2 \quad \Leftrightarrow \quad \neg(\neg f_1 \vee \neg f_2)$

(A3) $f_1 \Rightarrow f_2 \quad \Leftrightarrow \quad \neg f_1 \vee f_2$

(A4) $(f_1 \Leftrightarrow f_2) \quad \Leftrightarrow \quad (f_1 \Rightarrow f_2) \wedge (f_2 \Rightarrow f_1)$

(A5) $F(g_1) \quad \Leftrightarrow \quad \text{true} U g_1$

(A6) $G(g_1) \quad \Leftrightarrow \quad \neg F(\neg g_1)$

Implicitly, the logical symbols are defined for path formulas π analogously. Most important, however, is the new path quantifier A and the linear time operator F. These abbreviations allow for more compact and understandable CTL formulas.

The Bachus-Naur form (BNF) can be used to denote possible symbols for valid state and path formulas and summarizes the basic CTL rules alongside with common abbreviations that have been introduced. Valid state formulas f are described by the BNF in equation (2.1).

$$f ::= \top \,|\, \bot \,|\, a \,|\, \neg f \,|\, f \vee f \,|\, f \wedge f \,|\, f \implies f \,|\, f \Leftrightarrow f \,|\, E(g) \,|\, A(g) \tag{2.1}$$

In this context, $\top$ and $\bot$ are equivalent to true and false and can be seen as artificial propositions that do, respectively do not, hold in every state and on every path. Valid path formulas g are described by another BNF in equation (2.2).

$$g ::= \neg g \,|\, X(f) \,|\, f U f \,|\, F(f) \,|\, G(f) \tag{2.2}$$

2.2.2.2. Operator Semantics

The previous section introduced the syntax and formal semantics of CTL formulas which effectively consist of logical symbols, atomic propositions, linear time operators and path quantifiers. Linear time operators had to be prefixed with path quantifiers to allow for a valid CTL formula, resulting in 8 possible combinations of the linear time operators X, U, F and G with the path quantifiers E and A to valid CTL operators. This section tracks down each operator's definition, provides an informal description and a visualization based on state trees shown in figure 2.7. Keep in mind, however, that the semantics of CTL were defined with respect to Kripke structures with infinite sequences of states. State trees with finite traces are used for simplicity and demonstration purpose only.

EX The combination of the path quantifier E with the linear time operator X defines the CTL operator EX which is formally described by the rules 4 and 8 in equation (2.3).

$$\begin{aligned} &(R4) \quad s \models EX(f) \Leftrightarrow \exists \pi : (\pi_0 = s \wedge \pi \models Xf) \\ &(R8) \qquad\qquad\quad\; \Leftrightarrow \exists \pi : (\pi_0 = s \wedge \pi^1 \models f) \end{aligned} \tag{2.3}$$

Informally, $EX\,p$ holds when p holds in at least one of the possible immediate successor states. Figure 2.7a visualizes the CTL formula $EX\,p$. It shows a tree of states where states are linked to immediate successor states from lower levels. States in which the atomic proposition p holds are highlighted in blue, states where the CTL formula $EX\,p$ holds are circled in green. Graphically, $EX\,p$ holds whenever there is an immediate successor where p holds.

AX The counterpart of EX is AX, where each path has to fulfill X. As shown in equation (2.4), this operator is defined by the same rules but in combination with abbreviation 1.

$$\begin{aligned} &(A1) && AX(f) \Leftrightarrow \neg E \neg X f \\ &(R4) && \Rightarrow s \models AX(f) \Leftrightarrow \neg \exists \pi : (\pi_0 = s \wedge \pi \models \neg Xf) \\ &(R8) && \Leftrightarrow \neg \exists \pi : \big(\pi_0 = s \wedge \neg(\pi^1 \models f)\big) \\ &(R6) && \Leftrightarrow \neg \exists \pi : (\pi_0 = s \wedge \pi^1 \not\models f) \end{aligned} \tag{2.4}$$

Informally, $AX\,p$ holds when there is no immediate successor state in which p does not hold. A basic formula using this operator is visualized in figure 2.7b. As displayed, $AX\,p$ holds whenever p holds in all immediate successors which is equivalent to the phrase above.

EU The binary linear time operator U is a bit more complex which can be seen in rule 9. When combined with the existential path quantifier, it results in the CTL operator EX, which is defined by the rules 4 and 9, as shown in equation (2.5).

$$\begin{aligned} &(R4) \quad s \models E f_1 U f_2 \Leftrightarrow \exists \pi : (\pi_0 = s \wedge \pi \models f_1 U f_2) \\ &(R9) \qquad\qquad\qquad \Leftrightarrow \exists \pi : \Big(\pi_0 = s \wedge \exists k \in N_0 : \pi^k \models f_2 \wedge \forall (j \in N_0 | j < k)(\pi^k \models f_1)\Big) \end{aligned} \tag{2.5}$$

This operator checks whether there is a possible sequence of states that eventually leads to a state in which one state formula holds while another state formula holds in between, i.e. from the current state to this target state. To visualize this binary operator, the tree now shows a second proposition q in figure 2.7c. States in which q holds are highlighted in orange. A gradient from blue to orange denotes states in which both p and q hold. Every state in which q holds trivially fulfills the U operator. States in which p holds may as well fulfill $E\,pUq$ if there is a path to the leaves of the tree where p holds until q eventually holds. In result, any path consisting solely of states where either p or q holds until q holds the first time fulfills the U operator, thus EU holds in each state with at least one of such paths.

AU This operator is formally defined by abbreviation 1, in addition to the abbreviations and rules that define EU. Equation (2.6) presents the full definition.

$$\begin{aligned}
&(A1) && A\,f_1\,U\,f_2 \Leftrightarrow \neg E \neg (f_1\,U\,f_2)\\
&(R4) && \Rightarrow s \models A\,f_1\,U\,f_2 \Leftrightarrow \neg\exists\pi : (\pi_0 = s \wedge \pi \models \neg(f_1\,U\,f_2))\\
&(R9) && \Leftrightarrow \neg\exists\pi : \Big(\pi_0 = s \wedge \neg\exists k \in N_0 : \pi^k \models f_2 \wedge \forall(j \in N_0 | j < k)(\pi^k \models f_1)\Big)
\end{aligned} \tag{2.6}$$

To put this formalization in words, $A\,f_1\,U\,f_2$ holds when there is no sequence of states where there is no state in which f_2 holds while f_1 holds in all states in between. Again, every state in which q holds trivially fulfills this formula. States in which p holds may still fulfill it, if there is no path on which either q never holds or p does not hold until this state.

EF The CTL operator EF is formally defined as in equation (2.7), based on abbreviation 5 combined with the rules 4 and 9.

$$\begin{aligned}
&(A5) && EF(f) \Leftrightarrow E(\text{true}\,U\,f)\\
&(R4) && \Rightarrow s \models EF(f) \Leftrightarrow \exists\pi : (\pi_0 = s \wedge \pi \models (\text{true}\,U\,f))\\
&(R9) && \Leftrightarrow \exists\pi : \Big(\pi_0 = s \wedge \exists k \in N_0 : n^k \models f\Big)
\end{aligned} \tag{2.7}$$

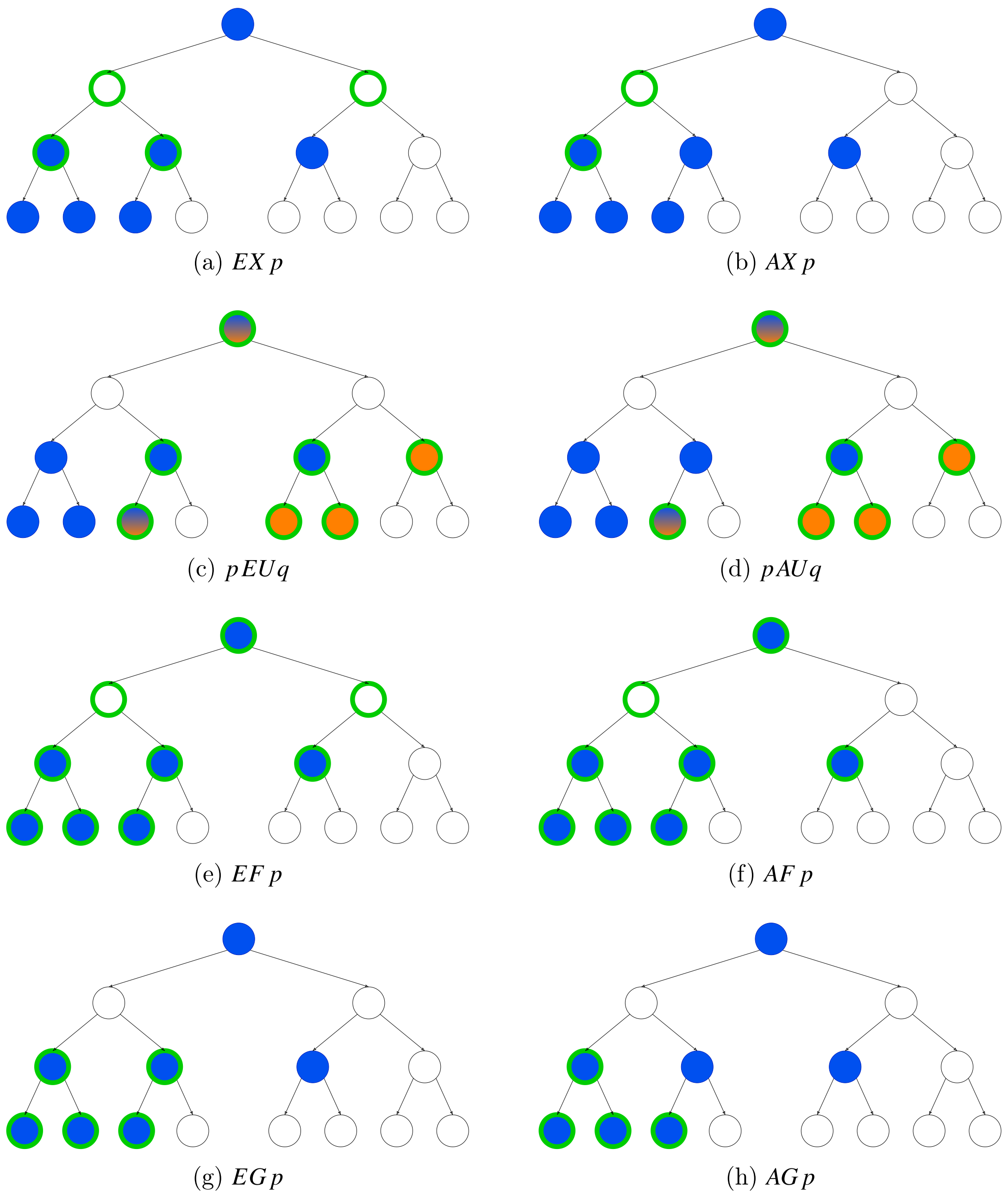
Informally, $EF\,p$ holds whenever there is at least one possible sequence in which p eventually holds. Graphically this translates to states where there is any path that contains any state in which p holds. As displayed in figure 2.7e, this is trivially the case for states where p holds but may still be true for other states if p holds in any subsequent state.

AF AF is defined by the rules 4 and 9 alongside with the abbreviations 1 and 5, as shown in equation (2.8).

$$\begin{aligned}
&(A1) && AF(f) \Leftrightarrow \neg E \neg F f\\
&(A5) && \Leftrightarrow \neg E \neg(\text{true}\,U\,f)\\
&(R4) && \Rightarrow s \models AF(f) \Leftrightarrow \neg\exists\pi : (\pi_0 = s \wedge \pi \models \neg(\text{true}\,U\,f))\\
&(R9) && \Leftrightarrow \neg\exists\pi : \Big(\pi_0 = s \wedge \neg(\exists k \in N_0 : n^k \models f)\Big)
\end{aligned} \tag{2.8}$$

Following this formalization, $AF\,p$ holds when there is no possible state sequence that does not include at least one state where p holds. Thus, each possible state sequence must include a state where p holds, This requirement is trivially fulfilled by states where p holds, as displayed in figure 2.7f. $AF\,p$ may still hold in other states if each possible path down to the leaves includes at least one state where p holds.

EG The CTL operator EG is formally defined by the abbreviations 6 and 5, allowing to



(a) $EX\,p$ (b) $AX\,p$

(c) $p\,EU\,q$ (d) $p\,AU\,q$

(e) $EF\,p$ (f) $AF\,p$

(g) $EG\,p$ (h) $AG\,p$

Figure 2.7.: Example formulas to visually describe the CTL operators. Circles represent states, edges represent state relations. Blue circles denote states where the atomic proposition p holds, green borders denote that the CTL formula holds. Orange circles denote states where q holds.

inductively define the model relation using the rules 4 and 9, resulting in equation (2.9).

$$\begin{aligned}
&(A6) & EG(f) &\Leftrightarrow E\neg F(\neg f) \\
&(A5) & &\Leftrightarrow E\neg(\mathrm{true}\,U\neg f) \\
&(R4) & \Rightarrow s \models EG(f) &\Leftrightarrow \exists\pi : (\pi_0 = s \wedge \pi \models \neg(\mathrm{true}\,U\neg f)) \\
&(R9) & &\Leftrightarrow \exists\pi : \Big(\pi_0 = s \wedge \neg(\exists k \in N_0 : n^k \models \neg f)\Big)
\end{aligned} \tag{2.9}$$

Informally, *EGp* holds whenever there is at least one possible sequence of states without any state in which *p* does not hold. To put it differently, *EGp* holds when at least one possible sequence consists solely of states where *p* holds. This is reflected by the visualization in figure 2.7g. States where *p* does not hold are the trivial case in which *G* can never hold. Instead, *EG* holds in states where (1) *p* holds and (2) there is at least one path down to the leaves with states where *p* holds all the time.

AG Formally, *AG* is defined by the abbreviations 1, 6 and 5, together with rule 4 as well as 9 this leads to the model relation definition shown in equation (2.10).

$$\begin{aligned}
&(A1) & AG(f) &\Leftrightarrow \neg E\neg G(f) \\
&(A6) & &\Leftrightarrow \neg E\neg\neg F(\neg f) \Leftrightarrow \neg EF(\neg f) \\
&(A5) & &\Leftrightarrow \neg E\,\mathrm{true}\,U\neg f \\
&(R4) & \Rightarrow s \models AG(f) &\Leftrightarrow \neg\exists\pi : (\pi_0 = s \wedge \pi \models \mathrm{true}\,U\neg f) \\
&(R9) & &\Leftrightarrow \neg\exists\pi : (\pi_0 = s \wedge \exists k \in N_0 : n^k \models \neg f)
\end{aligned} \tag{2.10}$$

AGp holds whenever there is no possible sequence of states, starting at the current state, with at least one state where *p* does not hold. In result, the operator holds when all possible state sequences solely consist of states where *p* holds. This property can be seen in figure 2.7h. *AGp* holds in states where (1) *p* holds and (2) *p* holds in every state on every path down to the leaves.

2.3. Graph Pattern Matching

Graph pattern matching is a field of research where graphs are searched for graphs (the pattern) and potential matches are retrieved. This matching procedure can be exact or approximate. Approximate graph matching tries to asses the similarity between the input graphs while exact graph matching retrieves mappings where the pattern structure is preserved in the searched graph (Gori et al. 2005). Exact graph matching can either retrieve the subgraph or supergraph of the pattern query (Sakr & Al-Naymat 2010). For supergraph queries, the searched graph is contained in the pattern graph while the first query type retrieves subgraphs of the searched graph which structurally match the pattern graph. In this case, the set of vertices and edges of the subgraph form a subset of the vertices and edges of the searched graph and the pattern graph is isomorphic to the subgraph. Approximate (or similarity) graph matching is not necessarily isomorphic (Sakr & Al-Naymat 2010).

A subgraph is defined as isomorphism if there is a bijective function mapping the vertices of the pattern graph to the subgraph of the searched graph (Sakr & Al-Naymat 2010). For each edge between two vertices in the pattern graph, there must be an edge between the vertices of the subgraph they have been mapped to. Both, the vertex and the edge mapping, has to respect the labeling of the respective graph elements. Formally, let $P = (V_P, E_P, \alpha_P, \beta_P, \delta_P, \varepsilon_P)$ and $S = (V_S, E_S, \alpha_S, \beta_S, \delta_S, \varepsilon_S)$ be pattern graph and subgraph, where V is the set of vertices, $E \subseteq V \times V$ is the set of edges, α, β are the sets of vertex respectively edge labels and $\delta : V \to \alpha$, $\varepsilon : E \to \beta$ are the labeling functions, assigning labels to vertices and edges. P is a graph isomorphism to S iff there is a bijective function $f : V_P \to V_S$ so that

1. $\forall (u,v) \in E_P : \exists (f(u), f(v)) \in E_S :$
2. $\alpha_P(u) = \alpha_S(f(u)) \wedge \alpha_P(v) = \alpha_S(f(v)) \wedge$
3. $\beta_P((u,v)) = \beta_S((u,v))$

This kind of graph pattern matching is generally computationally expensive. Many algorithms use dedicated indexing techniques to improve the complexity and indeed the performance heavily depends on the efficiency of these techniques and the query processing (Sakr & Awad 2010). Most techniques rely on a procedure with two phases, filtering and verification, where the first phase intensively uses indexing structures to reduce the state space (Awad & Sakr 2012, Wang et al. 2014, Shasha et al. 2002). Concerning the exact graph matching, there are mostly backtracking algorithms based on tree search (Gori et al. 2005, Wang et al. 2014).

Section 2.3.1 provides an overview about how graph pattern matching is used for BPQ and which graph-based query languages have been applied. Section 2.3.2 introduces DMQL, a generic graph querying language used as the reference language for graph-based BPQ.

2.3.1. Graph-based BPQ

Graph matching requires the model to be in a graph structure, having vertices and edges together with their properties (Riehle 2018). Arguably every conceptual model, like a BP model, already is such a graph (Weske 2012, Delfmann, Breuker, Matzner & Becker 2015, Riehle 2018). BP models, in particular, can be considered as graphs having behavior semantics (Wang et al. 2014). In principle model querying resembles the graph pattern matching problems of subgraph isomorphism and homeomorphism which can be applied to the graphs representing the respective model (Delfmann, Breuker, Matzner & Becker 2015). In the case of BPMN, such graphs are created by mapping flow objects (i.e. activities, events and gateways) to vertices. These vertices are linked by an edge if the respective flow elements are interlinked by connection objects. Properties of the flow and connection objects can be mapped directly to properties of the particular property graph elements. If the graph is not a property graph, each property key is mapped to a vertex type where the captions of its vertices denote the property value, for example. However, the evaluation of

BPQ queries is more than subgraph matching on the resulting graph (c.f. (Sakr & Awad 2010), (Beeri et al. 2006)). For example, graph patterns typically do not cover paths of arbitrary length (Delfmann et al. 2010).

The Business Process Query Language (BP-QL) is a visual language to retrieve paths from process definitions (Beeri et al. 2006). BP-QL focuses on querying distributed BP models modeled in the Business Process Execution Language (BPEL) (Andrews et al. 2003) and e.g. supports zooming into remove processes (Beeri et al. 2008). Its querying relies on an abstract model representing a collection of processes (c.f. (Beeri et al. 2006)). BP-QL has been specifically developed and applied for BPQ (see (Beeri et al. 2008)). The Generic Model Query Language (GMQL) is a textual query language for conceptual models, such as process models, data models and arbitrary graph-based models (Delfmann, Steinhorst, Dietrich & Becker 2015). In GMQL the model graph is searched for subgraphs corresponding to a pattern query. It has been developed with an application within business process compliance management in mind, too. In fact, GMQL has been applied for compliance checking in the financial sector (Becker et al. 2011) and a generic, GMQL-based compliance checking approach emerged (Bräuer et al. 2013). The Diagramed Model Query Language (DMQL) is its successor, extending GMQL with a visual notation and adding features such as loop detection (Delfmann, Breuker, Matzner & Becker 2015, Polyvyanyy 2018). Being a very recent development, DMQL has not been used for BPQ to our best knowledge yet. The Visual Model Query Language (VMQL) is a language to query conceptual models like Unified Modeling Language (UML) using the elements of the respective modeling language (Storrle 2009). In (Störrle & Acretoaie 2013), VMQL has been used for BPQ on BPMN models from the insurance sector. Though BPMN-Q is consequently categorized as logic-based BPQ, it applies graph-reduction techniques based on structural matching to reduce the state space for the LTL model checking (Awad, Decker & Weske 2008).

Besides BP-QL all these languages are generic and can be applied to different conceptual models. DMQL is the successor of GMQL and thus only VMQL and DMQL are considered as a reference language for graph-based BPQ. We favor the more recent DMQL over VMQL which has been initially developed for UML diagrams. While DMQL has not been applied for BPQ yet, GMQL has been used for CC more than once. DMQL has powerful features such as negative paths and ships with a generic visual notation.

2.3.2. Diagramed Model Query Language (DMQL)

The Diagramed Model Query Language (DMQL) is a multi-purpose model query language to specify graph structures using diagrams (Delfmann, Breuker, Matzner & Becker 2015). It has been developed in response to the identification of a set of 10 requirements for model query languages which no available language met. These requirements included, but were not limited to, 1. express pattern vertex properties, 2. map query edges to model paths with certain properties and elements, 3. be applicable to view-spanning integrated conceptual models and 4. provide a graphical editor to model diagrammed patterns. DMQL is defined on a formalization of conceptual models as graphs, using set theory, and thus can be applied to any conceptual model, regardless of the modeling language or notation that is being used.

A summary of the set-based formalization of DMQL is presented in section 2.3.2.1. Arguably one of the most powerful features of DMQL are paths with its properties, such as the ability to filter paths depending on nested patterns (subpattern). Section 2.3.2.2 describes the semantics of patterns and subpatterns in greater detail.

2.3.2.1. Formalization of DMQL

DMQL is a generic language which requires a metamodel for each conceptual model that is to be queried. As such, there is a modeling language that defines how supported conceptual models look like. Conceptual models of different modeling languages are allowed to be integrated, i.e. they are allowed to share elements which can be instrumented in DMQL patterns. Patterns are graphs referring to types and labels of the queried conceptual models. While patterns support numerous properties of vertices and edges, (negative) paths and even subpatterns for path edges, they also allow the definition of global rules. Finally, patterns are mapped to the joint graph of conceptual models - with respect to defined global rules - to produce a number of mappings. These mappings are subgraphs which match to the described pattern. The different concepts of the described matching procedure will now be introduced and partially formalized, summarizing (Delfmann, Breuker, Matzner & Becker 2015).

Delfmann et al. define a modeling language as a tuple $L = (T_V, T_E)$ where T_V is a set of vertex types and T_E is a set of edge types. Edge types are further separated into directed and undirected edge types, such that $T_E = T_{ED} \cup T_{EU}$. While $T_{ED} = T_V \times T_V$, undirected edge types must link vertices of different types, i.e. $T_{EU} \subseteq \{(t_1, t_2) | t_1, t_2 \in T_V, t_1 \neq t_2\}$.

A conceptual model is a tuple $M = (L, V, E, \alpha, \beta, C, \gamma)$ and thus an instantiation of a modeling language (Delfmann, Breuker, Matzner & Becker 2015). In this context, L is the instrumented modeling language and determines available vertex and edge types plus their interrelations, as defined above. V and E are non-empty sets of vertices and edges of the model, where $\alpha : V \to T_V$ and $\beta : E \to T_E$ are functions that map vertices and edges to their respective type. In accordance with edge types, edges can be further separated into directed and undirected edges, i.e. $E = E_D \cup E_U$. Multi-edges are allowed, as such two vertices may be linked by multiple edges at the same time. While directed edges in $E_D \subseteq V \times V \times N$ may link vertices to any vertex, undirected edges can only link vertices of different types, formally $E_U \subseteq \{(v_1, v_2, n) | v_1, v_2 \in V, n \in N, v_1 \neq v_2\}$. Note that multi-edges are numbered using the set of natural numbers N, counting from 1. Finally, C is a set of captions (or labels) and $\gamma : Z \to C$ with $Z = V \cup E$ is a generic labeling function for all graph elements. Vertices may have further attributes which in turn are regarded as vertices.

Multiple conceptual models of different modeling languages using shared concepts are referred to as integrated models. They can be used to combine models from different subdomains in the context of vertical abstraction (Weske 2012), e.g. process and organization models. Using shared concepts implies not only that the languages have one or more vertex/edge types in common but that the models refer to some identical modeling elements, e.g. a certain organizational unit (Delfmann, Breuker, Matzner & Becker 2015). Formally, integrated models can be represented as unified models, created by applying the union operation on conceptual models which are to be integrated. As such, a set of integrated model is defined as $\bar{M} = (\bar{L}, \bar{V}, \bar{E}, \bar{\alpha}, \bar{\beta}, \bar{C}, \bar{\gamma})$ where $\bar{L} = \cup_i L_i$, $\bar{V} = \cup_i V_i$, $\bar{E} = \cup_i E_i$, $\bar{\alpha} : \bar{V} \to \bar{T}_V \in \bar{L}$, $\bar{\beta} : \bar{E} \to \bar{T}_E \in \bar{L}$, $\bar{C} = \cup_i C_i$ and $\bar{\gamma} : \bar{Z} \to \bar{C}$ with $\bar{Z} = \bar{V} \cup \bar{E}$.

Given these set-based definitions that formalize models as graphs, DMQL uses patterns to issue queries. Patterns are graphs which are mapped to subgraphs of the joint conceptual models graph. A pattern is a tuple $Q = (V_Q, E_Q, P_V, P_E, \delta, \varepsilon, \rho, G)$. V_Q and $E_Q = V_Q \times V_Q \times N$ are the sets of pattern vertices and edges. Each pattern vertex and edge can be assigned to properties stored as tuples in P_V or P_E, respectively. The two functions $\delta : V_Q \to P_V$ and $\varepsilon : E_Q \to P_E$ assign such property tuples to pattern vertices and edges. $\rho \subseteq \cup_i L_i$ is the set of modeling languages the query pattern supports. Any query that is evaluated on a conceptual model returns a set of pattern occurrences.

There are 3 built-in properties for pattern vertices: $P_V \subseteq \{(vid, vcaption, vtypes)\}$, where vid is a string identifier for pattern vertices, $vcaption \in C$ is a label required for model vertices to be mapped and $vtypes \subseteq T_V \in \rho$ is the set of vertex types a model vertex is allowed to have. In contrast, there are 10 built-in pattern edge properties: $P_E \subseteq \{(eid, ecaption, dir, minl, maxl, minvo, maxvo, mineo, maxeo, vtypesr, vtypesf, etypesr, etypesf, \Theta)\}$. Analogously to vertices, eid is a string identifier for pattern edges and $ecaption \in C$ is a label required for model edges to be mapped. A pattern edge may be either mapped to a model edge or a model path, depending on the values of $minl$ and $maxl$. In both cases, $dir \in P(\{org, opp, none\}) \cup \varnothing$ determines whether mapped model edges or paths should have either the same, the opposite, none, any combination of those or even any direction, with respect to the pattern edge. The property $minl \in N$ specifies the minimum and $maxl \in N_0$ the maximum length of a mapped model path. Pattern edges with $maxl = 0$ are mapped to model paths of any length. Pattern edges with $minl = maxl = 1$ will be mapped to model edges only. Additional built-in edge properties other than eid, dir and $etypesr$ will be ignored in this case. $minvo, mineo \in N_0$ and $maxvo, maxeo \in N_0 \cup \{-1\}$ define to what extent paths are allowed to cut themselves by specifying the minimum and maximum number of allowed vertex and edge overlaps. The maximum number might be unlimited as indicated by a value of -1. $vtypesr, vtypesf \subseteq T_V \in \rho$ and $etypesr, etypesf \subseteq T_E \in \rho$ specify vertex and edge types that are required or forbidden on the path. Finally, $\Theta \subseteq \{(Q, c)\}$ specifies sub-patterns that the path has to map to. This set consists of tuples where $c \in \{req, prep, forb, pforb\}$ steer whether the sub-pattern is (partially) required or forbidden and $Q \neq \varnothing$ is just another query pattern.

The last pattern element is a tuple of global rules, $G = (gminvo, gmaxvo, gmineo, gmaxeo, R)$, which are applied to the whole query. Similarly to the respective edge properties, $gminvo, gmineo \in N_0$ and $gmaxvo, gmaxeo \in N_0 \cup \{-1\}$ define to what extent different paths of the pattern are allowed to cut each other. R is a set of rules operating on the properties of pattern vertices, supporting primitives, logical operators and calculations. There are four element properties that can be used: The caption, the vertex type and the number of incoming, outgoing or undirected edges connected to the vertex. Naturally, all but the caption property only apply to vertices.

Finally, all these constructs are used to map the patterns onto (possibly integrated) conceptual models. Given a set of integrated conceptual models $\bar{M}$ and a query pattern Q as input, a query produces a set of pattern occurrences $M_i(Q)' = (L_i', V_i', E_i', \alpha_i', \beta_i', C_i', \gamma_i')$ where $L_i' \subseteq \bar{L}$, $V_i' \subseteq \bar{V}$, $E_i' \subseteq \bar{E}$, $\alpha_i' = \bar{\alpha}|_{V_i'}$, $\beta_i' = \bar{\beta}|_{E_i'}$, $C_i' \subseteq \bar{C}$ and $\gamma_i' = \bar{\gamma}|_{C_i'}$.

Auxiliary constructs are defined to keep the formal description of how patterns are mapped to conceptual models short: Properties of vertices and edges are represented as tuples $t = (x_1, ..., x_n)$ but a component x_i of t can be accessed via $t.x_i$ where $t.x_i = t * (y_1, ..., y_n)$ with $\forall (j \in [1,n] \,|\, j \neq i) : y_j = 0$ and $y_i = 1$. A path through the set of models $\bar{M}$ is defined as a sequence of vertices and edges $< v_1, e_1, v_2, e_2, ..., v_{n-1}, e_{n-1}, v_n$ where the edges may be directed or undirected. The construct *path* is introduced, to describe such sequences. As specified, pattern edges may be mapped to model edges/paths in different directions, depending on the value of *dir*. Formally:

$$\begin{aligned} \text{path}(v_s, v_t)_{dir} = < v_s, e_s, v_{s+1}, e_{s+1}, ... v_{t-1}, e_{t-1}, v_t >, v_i \in \bar{V} \wedge \\ (\{org\} \in dir \wedge e_i = (v_i, v_{i+1}, z) \in \bar{E} \, \forall i \in [s, t-1], z \in N) \vee \\ (\{opp\} \in dir \wedge e_i = (v_{i+1}, v_i, z) \in \bar{E} \, \forall i \in [s, t-1], z \in N) \vee \\ (\{none\} \in dir \wedge e_i = \{v_i, v_{i+1}, z\} \in \bar{E} \, \forall i \in [s, t-1], z \in N) \end{aligned} \tag{2.11}$$

Since there may be more than one path between two vertices v_s and v_t, an index $k \in N$ is introduced to distinguish between the separate paths $\text{path}_k(v_s, v_t)_{dir}$. After the introduction of DMQL in 2015, features have been added. For example, DMQL has been extended by negative paths. They require that two existent vertices are not connected by a path with the given properties. However, patterns have to be fully connected. Thus vertices can not be connected to the pattern graph by negative paths, solely. Each vertex must still be connected by at least one positive path, given that there are pattern elements other than this vertex.

To determine which model elements are part of a path, two operations transforming a path to a set of vertices or edges, respectively, are described, given path $p = \text{path}_k(v_s, v_t)_{dir}$:

$$\text{vertices}(p) = \{v \in \bar{V} \,|\, v \in p\} \tag{2.12}$$

$$\text{edges}(p) = \{e \in \bar{E} \,|\, e \in p\} \tag{2.13}$$

Additionally, constructs to denote the path length as well as the number of overlapping vertices and edges on a path are defined:

$$\text{length}(p) = |\text{vertices}(p)| - 1 \tag{2.14}$$

$$\text{numvo}(p) = \lceil \frac{|p|}{2} \rceil - |\text{vertices}(p)| \tag{2.15}$$

$$\text{numeo}(p) = \lfloor \frac{|p|}{2} \rfloor - |\text{edges}(p)| \tag{2.16}$$

With these constructs, the mapping of a query pattern Q onto a set of models $\bar{M}$ can be defined in a more succinct way. Evaluating a query $Q = (V_Q, E_Q, P_V, P_E, \delta, \varepsilon, \rho, G)$ on a set of models $\bar{M} = (\bar{L}, \bar{V}, \bar{E}, \bar{\alpha}, \bar{\beta}, \bar{C}, \bar{\gamma})$ by mapping Q to $\bar{M}$ given a label equivalence relation $\sim \subseteq \bar{C} \times \bar{C}$ produces a set of pattern occurrences $M_i(Q)' = (L_i', V_i', E_i', \alpha_i', \beta_i', C_i', \gamma_i')$ such that equation (2.17) (handling simple pattern edges) and equation (2.18) (handling path edges) hold.

$$v_{Qs}, v_{Qt} \in V_Q \wedge e_Q = (v_{Qs}, v_{Qt}, n_Q) \in E_Q \wedge \varepsilon(e).minl = \varepsilon(e).maxl = 1 \wedge dir = \varepsilon(e).dir \Leftrightarrow \tag{2.17a}$$

$$\begin{aligned} \exists^{=1} v_{is}' \in V_i', \exists^{=1} v_{it}' \in V_i' : \alpha_i'(v_{is}') \in \delta(v_{Qs}).vtypes \wedge \delta(v_{Qs}).vcaption \sim \gamma_i'(v_{is}') \wedge \\ \alpha_i'(v_{it}') \in \delta(v_{Qt}).vtypes \wedge \delta(v_{Qt}).vcaption \sim \gamma_i'(v_{it}') \wedge \end{aligned} \tag{2.17b}$$

$$\begin{aligned} ((\exists^{=1} e_{io}' = (v_{is}', v_{it}', n_M) \in E_i' : \varepsilon(e_Q).ecaption\ \gamma_i'(e_{io}') \wedge org \in dir \wedge \beta_i'(e_{io}') \in \varepsilon(e_Q).etypesr) \vee \\ (\exists^{=1} e_{ii}' = (v_{it}', v_{is}', n_M) \in E_i' : \varepsilon(e_Q).ecaption\ \gamma_i'(e_{ii}') \wedge opp \in dir \wedge \beta_i'(e_{ii}') \in \varepsilon(e_Q).etypesr) \vee \\ (\exists^{=1} e_{in}' = \{v_{is}', v_{it}', n_M\} \in E_i' : \varepsilon(e_Q).ecaption\ \gamma_i'(e_{in}') \wedge none \in dir \wedge \beta_i'(e_{in}') \in \varepsilon(e_Q).etypesr)) \end{aligned} \tag{2.17c}$$

Informally, a simple pattern edge from pattern vertex v_{Qs} to v_{Qt} maps to a model edge between a pair of model vertices v_{is}', v_{it}' iff

1. the model vertex types and captions fit the respective pattern vertex properties and
2. the model edge caption and direction match the specified pattern edge.

This is formally expressed in equation (2.18). Equation (2.18), on the other hand, handles path pattern edges between the two pattern vertices v_{Qs} and v_{Qt} (a). A model path maps to the path pattern edge iff the start and end vertices have the configured types and captions (b) and its length is appropriate (c). Furthermore, the number of vertex and edge overlaps between elements on the path must be within the configured interval (d) and each vertex must have an allowed type but none of the forbidden types (e), this applies to edges as well (f). Depending on the configured constraint of sub-pattern occurrences, specified sub-patterns are fully or partly required or forbidden (g). This is applicable for each model i and multi-edge count n_Q, each pair of start and target pattern vertex indices s and t, for each sub-pattern occurrence θ and each sub-pattern mapping. At the same time, the configured inter-path overlapping constraints must be fulfilled by the mapping (i) and all global rules must hold (j).

$$
\begin{aligned}
v_{Qs}, v_{Qt} \in V_Q \wedge e_Q = (v_{Qs}, v_{Qt}, n_Q) \in E_Q \wedge dir = \varepsilon(e).dir \wedge \\
\varepsilon(e).minl > 1 \wedge \varepsilon(e).maxl \neq 1 \Leftrightarrow && \text{(2.18a)}\\
\exists^{=1} v'_{is} \in V'_i, \exists^{=1} v'_{it} \in V'_i : \alpha'_i(v'_{is}) \in \delta(v_{Qs}).vtypes \wedge \delta(v_{Qs}).vcaption \sim \gamma'_i(v'_{is}) \wedge \\
\alpha'_i(v'_{it}) \in \delta(v_{Qt}).vtypes \wedge \delta(v_{Qt}).vcaption \sim \gamma'_i(v'_{it}) \wedge && \text{(2.18b)}\\
p = \mathrm{path}_k(v'_{is}, v'_{it})_{dir}, \exists^{=1}\mathrm{vertices}(p) \in V'_i, \exists^{=1}\mathrm{edges}(p) \in E'_i : \\
l = \mathrm{length}(p), (\varepsilon(e_Q).maxl = 0 \vee l \leq \varepsilon(e_Q).maxl) \wedge \varepsilon(e_Q).minl \leq l \wedge && \text{(2.18c)}\\
o_V = \mathrm{numvo}(p), (\varepsilon(e_Q).maxvo = -1 \vee o_V \leq \varepsilon(e_Q).maxvo) \wedge \varepsilon(e_Q).minvo \leq o_V \wedge \\
o_E = \mathrm{numeo}(p), (\varepsilon(e_Q).maxeo = -1 \vee o_E \leq \varepsilon(e_Q).maxeo) \wedge \varepsilon(e_Q).mineo \leq o_E \wedge && \text{(2.18d)}\\
\forall t_V \in \varepsilon(e_Q).vtypesr \exists v \in \mathrm{vertices}(p) \,|\, \alpha'_i(v) = t_V \wedge \nexists v \in \mathrm{vertices}(p) \,|\, \alpha'_i(v) \in \varepsilon(e_Q).vtypesf \wedge && \text{(2.18e)}\\
\forall t_E \in \varepsilon(e_Q).etypesr \exists e \in \mathrm{edges}(p) \,|\, \beta'_i(e) = t_E \wedge \nexists e \in \mathrm{edges}(p) \,|\, \beta'_i(e) \in \varepsilon(e_Q).etypesf \wedge && \text{(2.18f)}\\
o_s = (\mathrm{vertices}(p) \cup \mathrm{edges}(p)) \cap M'_\theta(\Theta.Q) \wedge \\
((\Theta.constraint = req \wedge o_s = M'_\theta(\Theta.Q)) \vee \\
(\Theta.constraint = preq \wedge o_s \neq \varnothing) \vee \\
(\Theta.constraint = pforb \wedge o_s \neq M'_\theta(\Theta.Q)) \vee \\
(\Theta.constraint = forb \wedge o_s = \varnothing)) && \text{(2.18g)}\\
\forall i, n_Q, s, t, \theta, l, m \neq l, M'_l \neq M'_m \wedge && \text{(2.18h)}\\
\forall p \in M_i(Q) : \\
o_{Vg} = |\cap_k \mathrm{vertices}(p)|, G.minvo \leq o_{Vg} \wedge (G.maxvo = -1 \vee o_{Vg} \leq G.maxvo) \wedge \\
o_{Eg} = |\cap_k \mathrm{edges}(p)|, G.mineo \leq o_{Eg} \wedge (G.maxeo = -1 \vee o_{Eg} \leq G.maxeo) \wedge && \text{(2.18i)}\\
\forall r \in G.R : r = true && \text{(2.18j)}
\end{aligned}
$$

2.3.2.2. Pattern and Subpattern Semantics

As shown in section 2.3.2.1, DMQL patterns represent queries and consist of vertices and edges with their properties, potentially constrained by a number of global rules. This section assesses the semantics of these constructs, having an appliance in the BPQ context in mind.

Vertices in a pattern refer to (existing) model vertices. Pattern vertices are distinct within their pattern, i.e. two vertices in a pattern can not refer to the same model vertex. They feature three properties to be used: *vid*, *vcaptions* and *vtypes*. Firstly, *vid* is an identifier for a vertex within a pattern. It is solely defined within the scope of this pattern and does not imply any inter-dependencies when used in a second pattern. Thus, it is only use is to refer to vertices from global rules. Secondly, *vcaption* is a property to define an expression that mapped vertex labels have to match. It can be used to query specific activities or events but does not support the binding of variables, such as in VMQL or BPMN-Q. Thus, it is not suitable to refer to vertices from other patterns. Thirdly, *vtypes* is a property to define a set of allowed vertex types. It can be used to query particular types of vertices, such as events or activities.

Edges connect two vertices, either directly (simple edge) or with intermediate vertices allowed (path edge). Since their start and end vertices are distinct, edges are distinct within patterns too. They feature numerous properties, most prominently to define subpatterns for path edges. Analogously to *vid*, *eid* is an identifier for edges, valid within one pattern and used to refer to edges from global rules. Similarly, *ecaption* allows to define an expression that mapped edges' labels have to match. However, typically edges in BPMN do not feature a label (except for conditional sequential flow). The property *dir* provides a mean to specify the desired edge direction i.e. exactly as or inverse to as modeled in the pattern, without a direction or any combination of them is allowed. In the BPQ setting, one is usually interested in a single direction but sometimes it is useful to include multiple directions, e.g. when read or write access to data objects is to be fetched. The properties *vtypesr* and *vtypesf* allow to define sets of vertex types which are allowed, respectively forbidden, on the path. Note that the path contains both the start and the end vertex of the path, thus it is not possible to exclude their types from the path. In the BPQ context this prohibits to find the path between subsequent activities (i.e. without a third activity in between) using this property alone, for example. Similarly to these properties, *etypesr* and *etypesf* allow to define sets of allowed (or forbidden) edge types. However, the subpattern edge property (see below) can be used to filter out unwanted edge types, too. Thus *etypesf* is an abbreviation and does not provide additional features. The properties *minl* and *maxl* provide means to restrict paths to model paths of certain minimum or maximum length. While this can be used to query activities that are followed by another activity with at maximum n edges, intermediate events and branching or joining gateways have to be considered when specifying n. As, in theory, an arbitrary number of joining gateways may lie in between two activities, path lengths would only proof useful in a scenario where model graphs are free from gateways. Another set of edge properties is the minimum and maximum number of overlapping vertices and edges, namely *minvo*, *maxvo*, *mineo* and *maxeo*. Overlapping elements indicate loops, thus these properties are built-in features to detect loops, having the specified number of elements in common. The desired number of iterations can be specified via $minvo = maxvo = numIter * numLoopVertices$ where $numIter$ is the number of iterations and $numLoopVertices$ is the number of vertices inside the loop.

Finally, the subpattern property θ provides means to filter paths depending on nested patterns. Subpattern can use the same constructs as regular patterns and are either required or forbidden, partially or at a whole. In the DMQL query evaluation, matches for a subpattern are intersected with matches for the parental path, where the subpattern property was specified. Whether the subpattern is (partially) required or forbidden determines the allowed values for this intersection. The following examples refer to figure 2.8, where a pattern path was mapped to a model path $p = p_1 \to p_2$, and three subpattern paths were mapped to the model paths a, b and c. Vertices on the matched path p are highlighted by a light blue background.

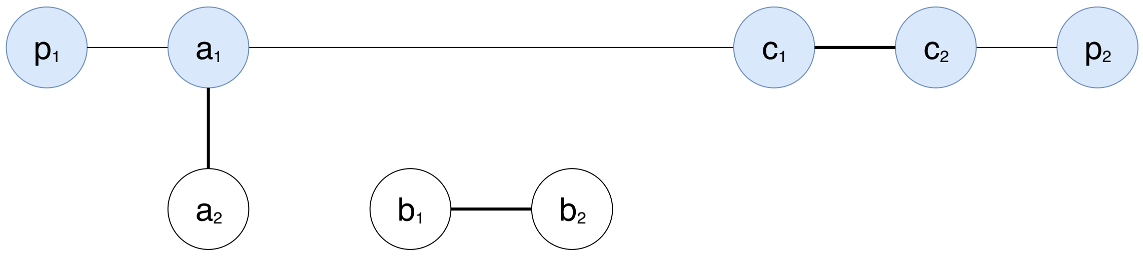


Figure 2.8.: Semantics of subpattern matches (a, b, c) with respect to the parental path match p.

If a subpattern is required (*req*), all the model elements it maps to have to be on the parental path. Only model path $c = c_1 \rightarrow c_2$ fulfills this requirement. Thus p would be a match if the subpattern which has been mapped to path c was required. When specifying a path as a required subpattern, this path may be any sub-path of the parental path. By default, using the same *vid* in the parental path and its subpattern does not imply that the respective pattern vertices refer to the same mode vertices. However, using domain knowledge such as the uniqueness of start and end events on a single BP model path, we can use vertex properties to make them refer to the same vertices. Another option is to incorporate global rules to tie the start and end vertices of the parental and the subpattern path together, i.e. require them to start and end with the same vertices. Global rules are discussed and applied in section 3.3.5.

If a subpattern is partially required (*reqp*), any of its mapped model elements has to be on the parental path. Model paths c and $a = a_1 \rightarrow a_2$ both fulfill this requirement, as c lies on p and so does a's vertex a_1. Thus p would be a match if the subpatterns mapped to a and c were partially required. In the case of path a, please note that a_1 does not necessarily needs to be a modeled vertex, it is a vertex of the model path which the subpattern path has been mapped to. Thus a_1 may be a vertex on a path edge between two (modeled) vertices of the subpattern. Modeling a_1 as a dedicated vertex allows to apply global rules and, again, enforce a_1 to match a specific vertex of a parental path or limit a_1 to desired vertex types, captions, etc. Otherwise, a_1 can be any vertex of the model path.

If a subpattern is partially forbidden ($forbp$), any of its mapped model elements have to be outside of the parental path. As such, it is the negation of *req*. Both a and $b = b_1 \rightarrow b_2$ fulfill this requirement, as b lies outside of p and so does a's vertex a_2. Thus p would be a match if the subpatterns mapped to a and b were partially forbidden. Please note that a_2 does not necessarily needs to be a modeled vertex or feature a specific type (see above).

Finally, if a subpattern is forbidden entirely ($forb$), all its mapped model elements have to be outside of the parental path. As such, it is the negation of *reqp*. Consequently a single common element, e.g. an activity, gateway or event, would invalidate the path. Only b fulfills this strict absence requirement, thus p would still be a match if the subpattern that mapped to b was forbidden. Using the same technique as for required subpatterns, we can model a path which only maps to the parental path, if any. This technique is used for logically combining subpatterns, as presented in section 3.3.4.

3. Mapping the Features of CTL and DMQL

In this section we show how CTL formulas can be mapped to DMQL patterns and vice versa. These languages are exemplarily used to compare the features of temporal logic and graph pattern matching in the context of BPQ. To follow and understand presented mappings, chapter 2 provides the relevant background, introduces the syntax and discusses the semantics of CTL and DMQL in detail. Section 3.1 describes how BPs are represented and thus can be queried with the particular language. The two languages were developed for different purposes in different eras and areas of computer science and represent BPs differently. Section 3.2 points out constitutive differences which must be considered when mapping their capabilities. We present the mapping of CTL to DMQL in section 3.3, i.e. DMQL patterns that reconstruct the different components of CTL formulas. Analogously, CTL formulas mimicking the features of DMQL patterns are introduced in section 3.4. The findings of this chapter will be summarized and discussed in chapter 4.

3.1. Preparations for BPQ

While section 2.2.1 and section 2.3.1 describe how BPQ is conducted using temporal logic and graph pattern matching in general, this section briefly discusses concrete steps for CTL and DMQL. It presents how each language encodes BP models for querying and discusses the implications of the differences in encoding.

As described in section 2.1, BP models consist of elements - such as activities, events and data objects - plus their relationships. Both elements and relationships can incorporate numerous properties, e.g. captions. For a temporal logic like CTL, BP models have to be transformed into finite state systems: Activities and events become states while their properties become atomic propositions (Smith et al. 2012). Start event vertices map to initial states while end event vertices represent the final states. Data objects can also be incorporated in CTL (c.f. (Awad et al. 2011)). Data is generally encoded in system states directly and used to evaluate the space of possible states. However, data objects could also be modeled as dedicated states if required. In case relationships between elements (such as control and data flow) use properties that should be targeted by queries, relationships would have to be represented as states with propositions, too. For the sake of simplicity we assume that properties of model edges are not incorporated in queries. As a result, only BPMN flow objects (i.e. activities and events) are mapped to states. In DMQL the whole BP model is loaded as a graph which it already constitutes. Activity and event vertices are referred to as state vertices since these map to the states in the state machine for CTL. Note that DMQL is a generic query language and requires a meta model to work with a particular modeling notation. Appendix A presents a basic meta model to use DMQL on BPMN (see section 2.1.1), an industry standard for BP models. This meta model does not claim to be complete but covers the features required for the described BPQ.

BPQ, in the way defined, is generally interested in possible executions of a BP given its model alone. Executions of BP models form finite sequences of BP elements, namely activities and events. In our setting each execution will start and end with an event. Note that this is a convention, dedicated start and end events are optional in BPMN (Weske 2012). Traces in CTL, however, are infinite. When finite systems have to be represented as Kripke structures, infinite repetition of the system's last state is a common step (c.f. (Pesic 2008)). In fact, it ensures the well-defined meaning of temporal operators like G (Valmari 1996). Thus any trace will be suffixed by a repeating final state s_f. Condensing this suffix to a single occurrence of s_f produces a finite sequence of states which can be directly mapped to the paths used in DMQL. Those literally represent the finite sequences of BP elements by nature.

3.2. Semantic Differences

When mapping CTL to DMQL, it is required to stress out fundamental differences in underlying concepts. CTL is a temporal logic to construct formulas where systems are evaluated against these formulas by a model checker. The language has been developed for the verification of finite state machines. DMQL is a generic graph query language to formulate graph-based queries, here graphs are tried to be mapped to these queries via pattern matching by a query evaluation engine. As such, constitutional differences include which elements are analyzed (target object), how the evaluation of these elements is processed (evaluation objects) and how its result is represented (evaluation result). Mapping these concepts at forehand will increase their understanding and establish a context for mapping CTL formulas onto DMQL queries and vice versa.

At first and as stated above, CTL analyzes systems and was initially developed to verify systems and their properties. Systems are represented as finite Kripke structures. These consist of all valid states of the system, including variables and assigned values. Graph pattern matching works on directed graphs in DMQL. The graph elements are vertices and edges connecting these vertices. When comparing CTL with its systems to DMQL with its graphs, states and graph elements have to be mapped. Section 3.1 shows how both states and graph elements are mapped to BPMN elements to allow for BPQ.

Secondly, different target objects require different evaluation processes and objects. CTL is evaluated with respect to Kripke structures that contain the system states. While these states represent single points in time of the system's execution, CTL formulas can target future states by incorporating temporal operators such as EF. Whether a formula holds in a state depends on the states which can (not) be reached in the future. In CTL, states can have multiple next states to choose from. This branching is a core feature of CTL and it is not offered by every temporal logic, as described in section 2.2. Though the number of states is limited, they are analyzed as traces, infinite sequences of states which express valid executions of the system, starting in the respective state. In contrast, pattern matching searches for paths along graph elements that fulfill a particular query. Paths are finite sequences of graph vertices and their interlinking edges. While traces are infinite, section 3.1 describes how they can represent an execution of a business process and can be mapped to the finite paths.

Finally, the evaluation processes of CTL and DMQL yield different types of values. CTL formulas are evaluated to Boolean values, being either true or false. Each state in a system can be tested against a state formula. Whether a system fulfills a CTL (state) formula depends on its initial states. The final result will be whether all its initial states fulfill the CTL formula. If the CTL operator is existential (i.e. features an existential path quantifier), it looks for a trace which fulfills its inner linear time statement. One valid trace is sufficient to make the formula true. If the CTL operator is universal, however, no trace is allowed to violate the statement. This is manifested in abbreviation A1 as presented in section 2.2.2.1: $A(p) \Leftrightarrow \neg E(\neg p)$. DMQL queries, on the other hand, are mapped to graph and yield subsets with mapped graph elements. Such mappings consist of multiple elements whenever the query does. Additionally, a single model may feature a large number of mappings. Thus, the result of the evaluation of DMQL queries is a set of sets of graph elements (i.e. model elements). The set of mappings will be empty only if no subset of graph elements was able to fulfill the query. Both, results of existential and universal operators, can be mapped to DMQL: For existential formulas, non-empty mappings for queries mimicking the formulas' linear statement equal to an evaluation of the formula to true. Formally, given an existential formula F_E and an equivalent existential DMQL query Q_E, $F_E = true \Leftrightarrow M(Q_E) \neq \varnothing$ - with respect to the different semantics between traces and paths. For universal formulas, however, this does not apply. Since the number of total paths is unknown it is not feasible to query and count desirable (i.e. non-violating) paths to check whether there are (violating) paths left. Instead, it is more reasonable to query violating paths and require that no mapping is found for them. Given an universal formula F_A and an equivalent universal DMQL query Q_A thus $F_A = true \Leftrightarrow M(Q_A) = \varnothing$.

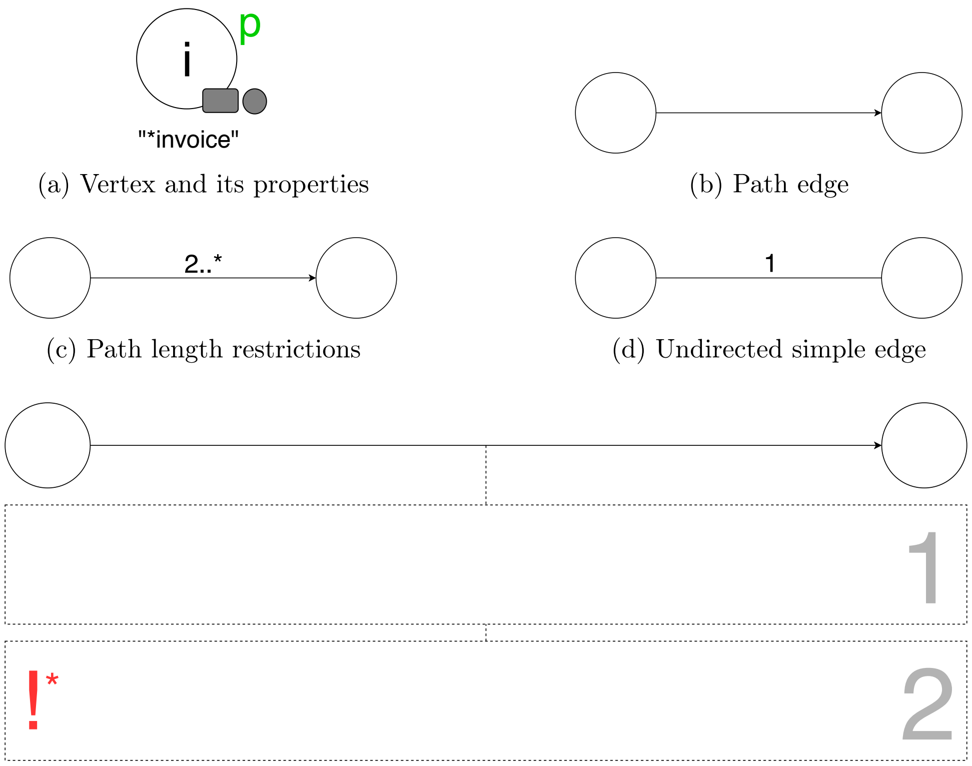
To conclude, CTL evaluates given formulas and produces binary results whereas pattern matching returns graph elements (i.e. graphs) that were successfully mapped to a given pattern graph. While traces are infinite sequences of states and paths are finite sequences of graph elements, both can be mapped to the finite sequences of model elements that the execution of a BP involves. In this context, CTL states can be mapped to activity and event vertices in DMQL.

3.3. Mapping CTL to DMQL

This section presents numerous DMQL queries that reconstruct the features of CTL. At first, section 3.3.1 introduces a customized visual notation for DMQL that was optimized to present the mapping queries. section 3.3.2 uses this notation to describe a basic structure and discusses its implications for the following queries. CTL primarily consist of its eight operators which can be nested and logically combined. DMQL queries for the basic operators are presented in section 3.3.3 while section 3.3.4 shows how operator queries can be logically combined. Finally, section 3.3.5 describes how they can be nested into each other to reconstruct complex formulas.

3.3.1. Customized Graphical Notation for DMQL

This section presents the customized graphical notation for DMQL which is used for queries in this thesis. As stated, DMQL ships with a dedicated graphical representation which is as generic as DMQL. For the sake of readability, it hides away certain attributes that could render the visualization of larger models confusing. For example, it presents subpatterns of a path edge in a subform. However, the developed mapping queries will make extensive use of subpatterns, as described in section 3.3.2. They will contain the main logic and thus need to be visible at any time. Additionally, vertex type shapes are hidden if multiple types are allowed for a certain pattern vertex. As the mappings use at maximum two vertex types at a time, they still can and should be shown. Though it is possible to display all the subforms of the built-in language next to the respective pattern, a custom notation can yield more concise representations.



(a) Vertex and its properties (b) Path edge

(c) Path length restrictions (d) Undirected simple edge

(e) Required and forbidden subpatterns

Figure 3.1.: Specialized visual notation for DMQL patterns.

Pattern vertices are presented as circles in our customized notation and may be enriched by elements to indicate their properties, as shown in figure 3.1a. Per default, a pattern vertex is mapped to model vertices of any type, i.e. property $vtypes = \varnothing$. If a model vertex is required to have a certain type, the shapes of allowed types will be shown in the lower right corner of the pattern vertex, much like a subscript. These shapes refer to the respective BPMN element types, namely circles for events and rounded rectangles for activities. Events can be start events, end events or intermediate events, for example, while activities may be atomic tasks or more complex such as subprocesses. If the pattern vertex refers to a specific event or activity type, the respective BPMN shape will be used. Otherwise, the vertex type indicator is the filled BPMN shape of the respective element type, as shown in the figure. The *vid* property of the vertex, if set, is placed inside the vertex circle while the caption is placed below. Predicates that have to hold in a state vertex will be placed in the upper right corner of that vertex. The font color of the predicate is optional and guides the viewer by indicating positive (e.g. q) and negated predicate statements (e.g. $!p$). Additional vertex properties, however, are not visualized.

Arrows represent both simple edges and path edges. Per default, there are no length restrictions and arrows represent paths of arbitrary length (see figure 3.1b). If there are lengths restrictions, they are placed inside a label above the arrow. For example, figure 3.1c shows a path with minimum length 2 and an arbitrary maximum length. Simple edges are path edges with a length of 1, i.e. this is both the minimum and maximum length, as indicated by the single number in figure 3.1d. The heads of the arrows indicate the direction of the pattern edge. In our context, there is usually one head which is pointing from the start to the end vertex. However, the heads may also be omitted (as in case of the simple edge) to represent undirected pattern edges. The subpattern of a path will be displayed in a dashed rectangle, connected to the respective path edge with a dashed line (see figure 3.1e). For unambiguous referring, subpatterns feature a label in their upper right corner. If a path edge has multiple subpatterns, the rectangles are chained visually. Per default, subpatterns are required at a whole (*req*). An exclamation mark at their left border indicates that the subpattern is forbidden to occur at a whole ($forbp$) instead. If single (but not all) elements of the subpattern are required or forbidden, an asterisk is added.

3.3.2. Basic Pattern Structure

This section introduces the basic structure of the DMQL patterns that were produced to reconstruct the CTL operators. The basic structure consists of a single path from a vertex s to a vertex e with a subpattern P, as shown in figure 3.2. The start vertex s refers to the starting point of the respective CTL operator. As described, the evaluation of operators can start in any state, depending on whether they are used at the formula's root level or nested in additional operators. Consider the following example formula to derive the implications for the starting point:

$$EF(AX(EX(Stopped))) \tag{3.1}$$

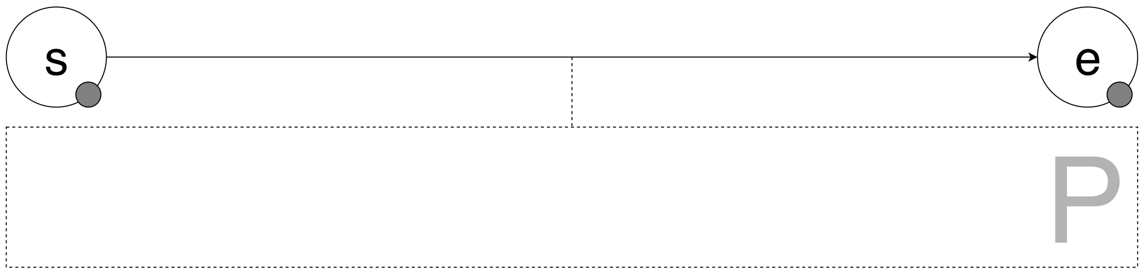


Figure 3.2.: Basic structure for the DMQL mapping patterns.

This formula consists of three nested formulas, namely $EF(x)$, $AX(y)$ and $EX(z)$. In this example $EF(x)$ is a root formula which is always evaluated against the entire system, i.e. all its initial states. $AX(y)$ is an inner formula which is evaluated against all states that the outer root formula has filtered. Since the outer formula operator is EF, these states are the same as for the outer formula. In contrast, the outer formula operator of $EX(z)$ is AX. Thus EX is evaluated in all states that directly follow the states which AX has been evaluated in. To subsume, the formula context and the exact nesting of the operators determine whether the evaluation states are the initial or subsequent states.

For patterns that reconstruct the basic operators, we assume that the respective operators are used at the formula's root level, i.e. their starting point is an initial state. In terms of DMQL, patterns thus have to start from a start event vertex in our BPQ setting. Start event vertices can be easily identified if they feature a respective property. We define the property $evPosType = \{start|intermediate|end\}$ (event position type) for event vertices that describes whether the vertex represents a start, intermediate or end event. s thus is a vertex of the type *event* where $s.evPosType = start$. In the pattern this is indicated by using BPMN's start event shape for the vertex type. In addition to this start vertex, the patterns feature a path of arbitrary length to a vertex e which always refers to an end event vertex. Consequently e is a vertex of the type *event* where $e.evPosType = event$, as indicated by using BPMN's end event shape for the vertex. This structure would not be necessary for every pattern and requires the existence of dedicated start and end event vertices. However, these are trivial to add in case they are missing in a model graph. Conveniently, the described setup guarantees a path to be existent. This allows to use the *subpattern* property of path edges, one of the most powerful tools in DMQL. Subpatterns can be used to refine the set of valid paths. In the presented structure, the subpattern P will express the operator and effectively limit the paths mapped by the pattern edge to paths that fulfill the operator. Note that subpatterns of the modeled path can refer to s and e as these are the only start and end event vertices that lie on the path. From operator pattern to operator pattern the basic structure is the same but the subpattern changes. Given an existential pattern Q_E where $M(Q_E) \neq \varnothing$ is desired, the respective universal pattern Q_A can be produced by negating the subpattern of Q_E. In this way, the new pattern Q_A filters paths that do not fulfill the original operator and if $M(Q_A) = \varnothing$, the model is shown to be free of such violating paths.

To summarize, following patterns consist of a single path of arbitrary length from a start event to an end event vertex. From pattern to pattern, the *subpattern* property of this edge changes to reconstruct the respective operator. This structure allows for a convenient conversion between existential and universal patterns.

3.3.3. CTL Operators

The following subsections present how DMQL queries can reconstruct the eight basic CTL operators, namely EX, AX, EU, AU, EF, AF, EG and AG. Basic, in this context, refers to the appliance of the operator to atomic propositions, i.e. without logically combining (see section 3.3.4) or nesting (see section 3.3.5) the operators. As a result, the operators will always be at the root level of a formula. For each operator, we discuss its semantics in the context of CTL and transfer it to DMQL's graph setting. This allows to express a DMQL query which equals the respective CTL operator, using the presented visual notation. For the sake of completeness, a formalization of these queries can be found in the appendix.

3.3.3.1. EX

Usually the CTL operator EX (section 2.2.2.2) can be used in any state to express that a certain predicate holds in at least one of the consecutive states. The term consecutive explicitly requests the absence of any states between the former and the latter state. As described, we assume that the operator is evaluated in an initial state here. In terms of paths there has to be a path from a start event vertex to a consecutive state vertex, in which the named predicate holds. Consecutive, however, does not necessarily imply that the path length is 1. Activities and events may be linked to gateways that split and join the control-flow in BPMN, for example. Thus there may be an arbitrary number of vertices on the path as long as there is no additional state vertex between the path's start and end vertices.

As described in section 3.3.2, DMQL queries will always start with a path of arbitrary length from the start event vertex s to an end event vertex e. They are identified by their types, indicated by the BPMN shapes. The arbitrary path length is realized via the path edge properties $minl = 1$ and $maxl = 0$ (which are the default values and thus not visualized). Again, to this point all queries are the same. Specific for queries is the subpattern of the described path edge which restricts the paths to those fulfilling the mimicked CTL operator. The DMQL pattern to reconstruct EXp is visualized in figure 3.3. EXp can be expressed by a path of arbitrary length from s to a state vertex i in which predicate p holds (subpattern 1). Latter state vertex is identified by its vertex type, as indicated by the subscript shapes. Each path either fulfills the subpattern or does not. If at least one path fulfills the subpattern, there is a start event where EXp holds. Again, this subpattern path must not include additional state vertices. Thus, the nested subpattern 1A forbids it to fully contain paths that feature three state vertices - s, i and an undesired third exemplar. Note that the two pattern vertices with identifier s automatically will refer to the same model vertex as there will be exactly one start event vertex on a path. Formally, $EXp \Leftrightarrow M(Q_{EX}) \neq \varnothing$ where Q_{EX} is the presented DMQL pattern. The formalization of Q_{EX}

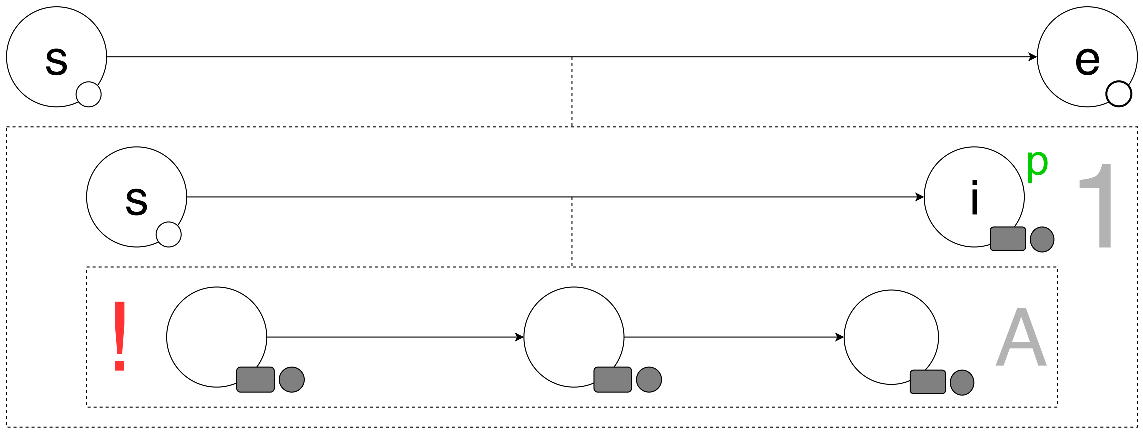


Figure 3.3.: DMQL pattern representing the CTL formula EXp.

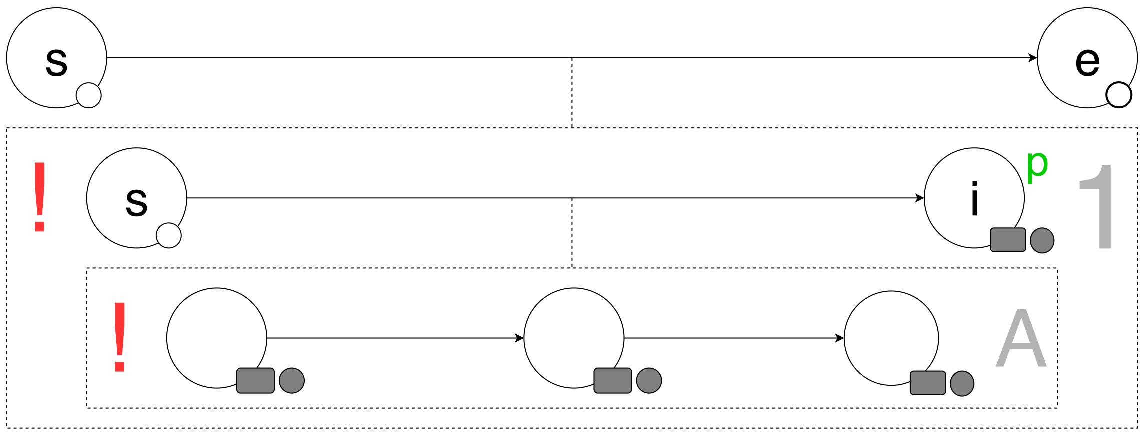


Figure 3.4.: DMQL pattern representing the CTL formula AXp.

is shown in appendix B.1, where subpattern 1 is formalized by $Q_{EX'}$ and Q_{NXTST} constitutes subpattern 1A.

3.3.3.2. AX

In contrast to EX, the CTL operator AX requires a certain predicate to hold in all consecutive states, not just only one. This is equivalent to $\neg EX \neg p$, stating that p holds in no consecutive state. In terms of paths: There is no path from a start event vertex to a consecutive state vertex, where EXp does not hold. As opposed to existential queries, $M(Q) = \varnothing$ is desired for universal queries. Thus, the pattern for AXp will look for paths violating the CTL formula and is shown in figure 3.4. Note that the pattern is the negated form of the respective existential pattern, i.e. for EXp (section 3.3.3.1). Again, all patterns are equal at the root level. Subpattern 1, however, is now forbidden instead of required. Hence the pattern looks for paths without an i that fulfills EXp. A single occurrence implies $EX\neg p \equiv \neg AFp$ holds. Its nested subpattern 1A is left untouched as its purpose still is to limit potential matches for vertex i to consecutive state vertices. As such, the query exactly looks for paths violating AXp now. Formally, $AXp \Leftrightarrow M(Q_{AX}) = \varnothing$ where Q_{AX} is presented in appendix B.2 with its subpatterns $Q_{AX'}$ (subpattern 1) and Q_{NXTST} (subpattern 1A).

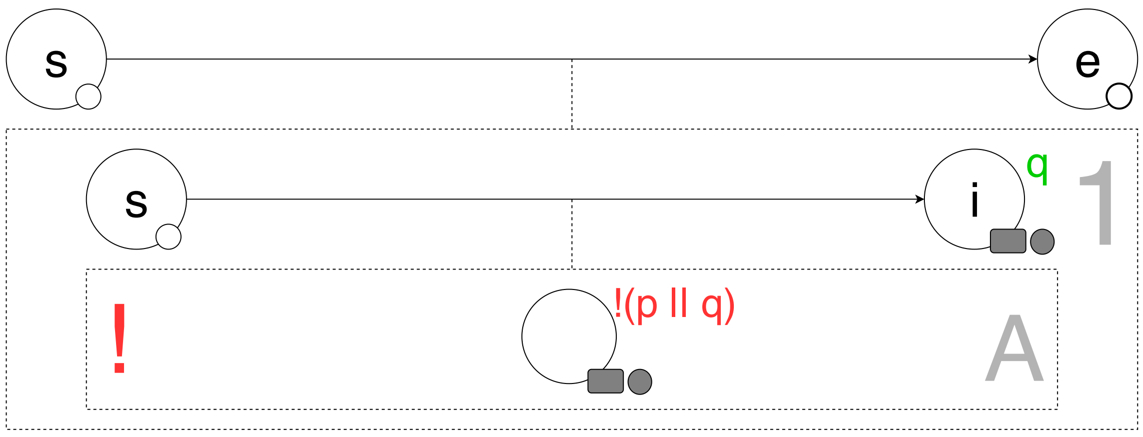


Figure 3.5.: DMQL pattern representing the CTL formula $pEUq$.

3.3.3.3. EU

The CTL operator EU can be used to express that a certain predicate p holds until, eventually, predicate q holds. The trivial case is that q holds in the initial state. Otherwise, there must be a state in which q eventually holds and p must hold in all intermediate states, including the initial state. As a first draft, there must be a path starting from the start vertex to a state vertex in which q holds, where there is no state vertex on that path in between in which p does not hold. This pattern would not cover the rare trivial case However, this could be solved by issuing a second query for the trivial case specifically. The translation of $pEUq$ to DMQL is presented in figure 3.5. Again, the root path edge from s to e is looking for full paths through the BP model. Using the subpattern property, mapped paths are required to contain a path of arbitrary length from a start event vertex to a state vertex i in which q holds (subpattern 1). Note that the subpattern implies $s \neq q$ and thus does not cover the trivial case, indeed. The subpattern path is required to be free of any state vertices in which neither p nor q holds (subpattern 1A). Including q in this condition is required as i will be checked against it, too, which is not required to fulfill p. Note that the sub-path does not stop at the first occurrence of a state vertex fulfilling q. However, additional occurrences do not invalidate EU. Formally, this pattern can be expressed as presented in appendix B.3, where there are two nested subpatterns and with $pEUq \Leftrightarrow M(Q_{EU}) \neq \varnothing$. Subpattern 1 is formalized by $Q_{EU'}$, whereas the subpattern 1A is covered by $Q_{\bar{p}}$.

3.3.3.4. AU

Analogously to EU, the AU operator requires that - on all future traces - there's a state reachable (1) in which a target predicate holds and (2) where a second predicate holds in all intermediate states (including the initial one). In terms of paths, a negative lookup is required again. Invalid paths either do not contain a state vertex in which predicate q holds or contain an intermediate state vertex in which predicate p does not hold. In DMQL, the path from s to e must not fulfill the criteria of EU at all. This violation occurs when it does not include the subpattern that would actually meet the operator's criteria.

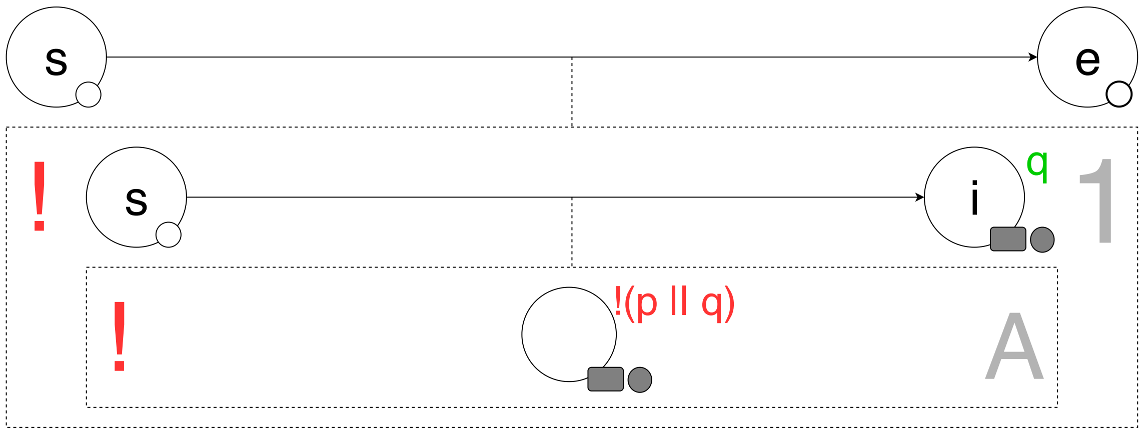


Figure 3.6.: DMQL pattern representing the CTL formula $pAUq$.

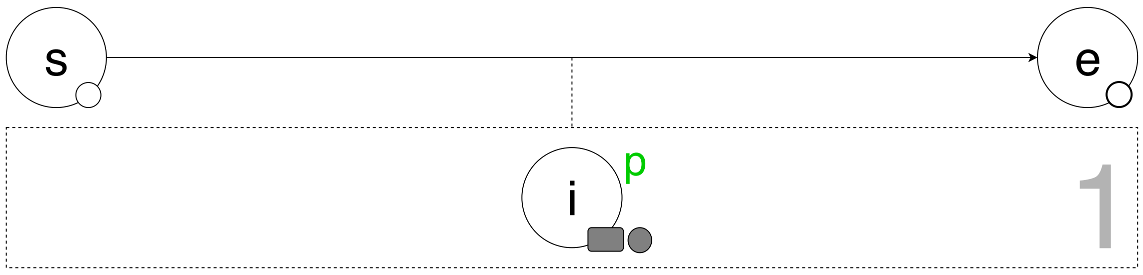


Figure 3.7.: DMQL pattern representing the CTL formula EFp.

As such, the subpattern is negated again, as visualized in figure 3.6. To verify the pattern, consider in which case no such sub-path (subpattern 1) would be existent: Either there is no sub-path to a state i at all or there is such a sub-path but it contains a vertex in which neither p nor q holds. This pattern can be formalized as shown in appendix B.4, where there are the named two nested subpatterns with $pAUq \Leftrightarrow M(Q_{AU}) = \varnothing$. Subpattern 1 is formalized by $Q_{AU'}$, whereas subpattern 1A is covered by $Q_{\bar{p}}$.

3.3.3.5. EF

The CTL operator EF states that a certain predicate holds in at least one future state on any trace, starting from the initial state. In the trivial case, the predicate holds in the initial state already. Expressed in terms of paths, the path from the start to the end event vertex contains a state vertex in which the predicate holds.

The respective DMQL pattern is presented in figure 3.7. Starting with the basic path edge again, a subpattern is used to filter paths that contain a state vertex i in which predicate p holds (subpattern 1). No additional elements are required, as both s and e may fulfill the formula (in contrast to EX). Formally, this pattern can be expressed as Q_{EF}, as in appendix B.5, where $EFp \Leftrightarrow M(Q_{EF}) \neq \varnothing$.

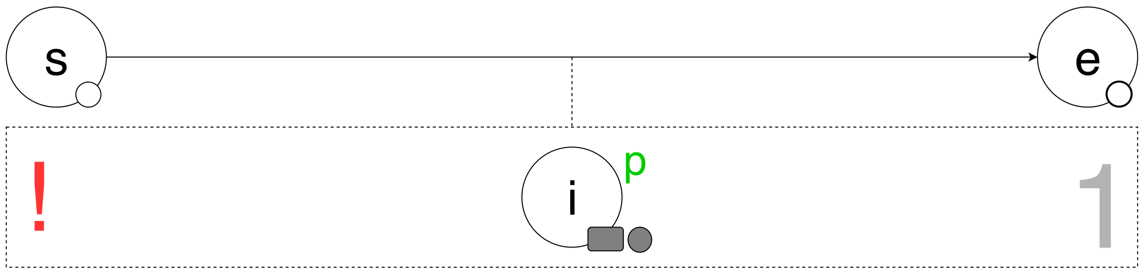


Figure 3.8.: DMQL pattern representing the CTL formula $AF\,p$.

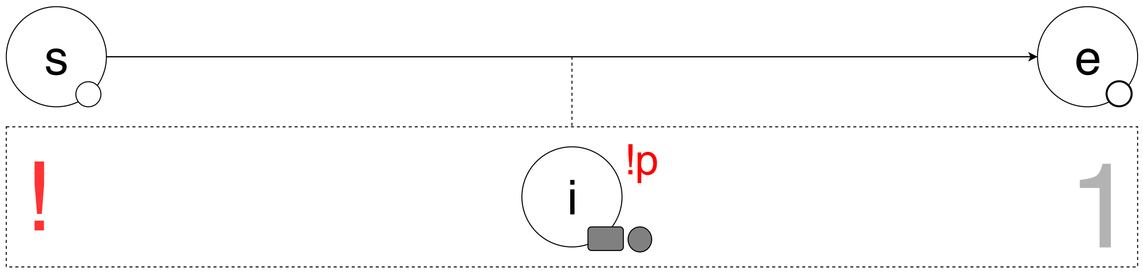


Figure 3.9.: DMQL pattern representing the CTL formula EGp.

3.3.3.6. AF

The AF operator can be used to filter states where every future trace contains a state in which a certain predicate holds. Equivalently, there is no trace with a future state reachable where the predicate does not hold. In terms of paths, there must not be a path from a start to an end event vertex that does not contain a state vertex where p holds. Analogously to AU, we negate the subpattern of EF as shown in figure 3.8: Subpattern 1 is now forbidden and thus the basic path solely maps to model paths that are free of state vertices in which p holds. In other words: model paths where EXp does not hold and which would clearly violate AXp. The formalization of this pattern is presented as Q_{AF} in appendix B.6, where $AF\,p \Leftrightarrow M(Q_{AF}) = \varnothing$ and $Q_{AF'}$ constitutes subpattern 1.

3.3.3.7. EG

The last existential operator, EG, expresses that there is a trace where a certain predicate holds in each of its states, starting from the initial state. Formally, there is no trace in which a future is reachable where the predicate does not hold. In path context, there is a path from a start to an end event vertex which is forbidden to include a state vertex in which the predicate is not fulfilled. Figure 3.9 visualizes the DMQL pattern for EGp. Subpattern 1 requires the path to be free of any state vertices in which p does not hold. Q_{EG} in appendix B.7 formalizes this pattern, where $EGp \Leftrightarrow M(Q_{EG}) \neq \varnothing$ and subpattern 1 is manifested in $Q_{EG'}$.

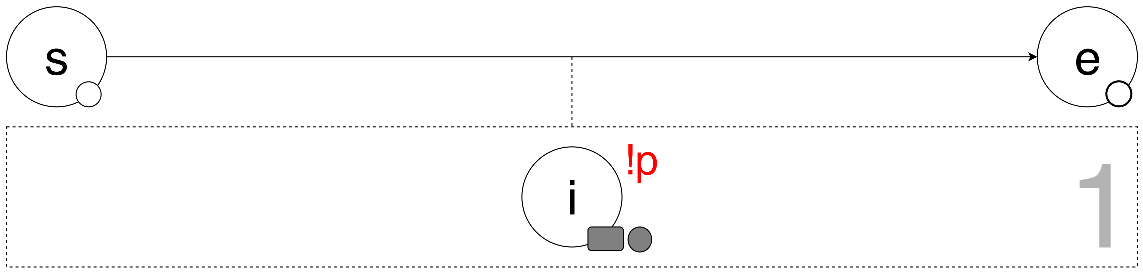


Figure 3.10.: DMQL pattern representing the CTL formula AGp.

3.3.3.8. AG

Analogously, AG requires each future trace to solely consist of states where a certain predicate holds. Thus, there has to be no path with a state vertex in which the predicate does not hold. Again, we negate the subpattern of the existential operator EG and get the pattern shown in figure 3.10. A basic path with a state vertex in which p does not hold clearly violates AGp. The respective formalization is presented in appendix B.8. Formally, $AGp \Leftrightarrow M(Q_{AG}) = \varnothing$ where $Q_{AG'}$ represents subpattern 1.

3.3.4. Logically Combining CTL Operators

Considering the BNF for state formulas (see section 2.2.2.1), CTL formulas may be combined in four different ways: Using the logical or ($\vee$), the logical and ($\wedge$), an implication ($\implies$) or an equality comparison ($\Leftrightarrow$). CTL operators are combined at root level as in $EFp \wedge EFq$ or in their nested form as in $EF(EFp \wedge EFq)$. Analogously, we separate between the logical combination of queries and the combination of nested subpatterns in DMQL.

3.3.4.1. Combining Root Operators

If CTL operators are combined via the logical and at the formula's root level, as in $EFp \wedge EFq$, both a state in which p holds and a state in which q holds must be reachable. The formula does not require the states to be on the same trace and it leaves open whether a single state or two distinct states fulfill the EF operators. In DMQL, we need to express the existence of two paths which may or may not refer to the same path. Thus, we can not simply connect two paths to a starting vertex and let them point at two vertices (see figure C.1). The latter were implied to be distinct, as they are within the same pattern. Instead, we could issue multiple queries, one for each operator, and combine their mapping results. Given two DMQL queries Q_{EF1} and Q_{EF2} that model the two operators (see section 3.3.3), EFp would hold if the mapping of Q_{EF1} is non-empty while EFq would hold if the mapping of Q_{EF2} is non-empty. Combined, the mapping results reconstruct the example formula:

$$EFp \wedge EFq = true \quad \Leftrightarrow \quad M(Q_{EF1}) \neq \varnothing \wedge M(Q_{EF2}) \neq \varnothing \tag{3.2}$$

The described mechanism works for an arbitrary number of operators at the formula's root level, regardless if they are existential or universal. Consider the CTL formula $EFp \wedge AFq$ as an example. Given two DMQL queries Q_{EF} and Q_{AF} that model the two operators, the mapping for the first is required to be non-empty while the second requires an empty mapping. This could not have been modeled within a single pattern, anyway. However, issuing multiple queries supports these different requirements and can set the formula equal to an appropriate combination of the queries' mapping results:

$$EFp \wedge AFq \quad \Leftrightarrow \quad M(Q_{EF}) \neq \varnothing \wedge M(Q_{AF}) = \varnothing \tag{3.3}$$

The described mechanism also works when combining root level operators using the logical or. In a single pattern, DMQL can not express that a path is optional let alone that n-out-of-m paths are required. However, the mapping results can be logically combined as specified in the mimicked CTL formula. Consider $EXp \vee AXq$ as an example. Given two DMQL queries Q_{EX} and Q_{AX} that model the two operators, combining the mapping results reconstructs the formula:

$$EXp \vee AXq \quad \Leftrightarrow \quad M(Q_{EX}) \neq \varnothing \vee M(Q_{AX}) = \varnothing \tag{3.4}$$

Able to reconstruct these two boolean operators, the mechanism can reconstruct the logical implication and the logical equality. Take $EFp \implies EFq$ as an example. Per definition, this CTL formula can be rewritten as $\neg EFp \vee EFq$ and thus multiple queries can reconstruct this formula as before. Given the two DMQL queries Q_{EF1} and Q_{EF2} again:

$$\begin{aligned}(EFp \implies EFq) &\equiv (\neg EFp \vee EFq)\\ &\Leftrightarrow M(Q_{EF1}) = \varnothing \vee M(Q_{EF2}) \neq \varnothing\end{aligned} \tag{3.5}$$

Note that EFp was negated, hence an empty mapping result is desired this time. Similarly, formulas as $EFp \Leftrightarrow EFq$ can be rewritten using solely the two mapped boolean operators and thus mapped to DMQL queries, by combining their results appropriately:

$$\begin{aligned}(EFp \Leftrightarrow EFq) &\equiv (EFp \implies EFq \wedge EFq \implies EFp)\\ &\equiv ((\neg EFp \vee EFq) \wedge (\neg EFq \vee EFp))\\ &\Leftrightarrow M(Q_{EF1}) = \varnothing \vee M(Q_{EF2}) \neq \varnothing) \wedge (M(Q_{EF2}) = \varnothing \vee M(Q_{EF1}) \neq \varnothing\end{aligned} \tag{3.6}$$

Again, the presented mechanism is solely intended for operators combined at the root level.

3.3.4.2. Combining Nested Operators

Operators are not combined at the root level exclusively. Instead, they may be nested and combined at arbitrary levels of the formula, such as in $EF(EFp \wedge EFq)$. While nested operators are not presented until the next section, this section already describes the mechanism to combine them. The mechanism assumes that each nested operator can be expressed as a subpattern and bases upon combining involved subpatterns into the appropriate path of the parental operator's query. This path will be denoted as $i \rightarrow e$ in the following queries. Note that this path implies $i \neq e$ so it can not be used to form queries that have an end event vertex as a starting point. However, this is a rare use case and could be achieved by issuing another query.

The logical and is the most simple logical connector to reconstruct. Consider the following example formula which contains two nested operators:

$$EF(EFp \wedge \neg EFq) \tag{3.7}$$

Assume that two subpatterns P and Q model the nested operators. When defining multiple subpatterns for a path, DMQL requires each to be present or absent, depending on whether it is required or forbidden. This naturally models the logical and as all of them have to match their defined criteria. Nested positive operators like EFp map to a required subpattern while every negated operator such as $\neg EFq$ maps to a forbidden subpattern. In fact, nested operators will be partially required or partially forbidden but this aspect will be covered in the following section that introduces the nesting of operators. Figure 3.11 visualizes a DMQL query that combines the two subpatterns appropriately.

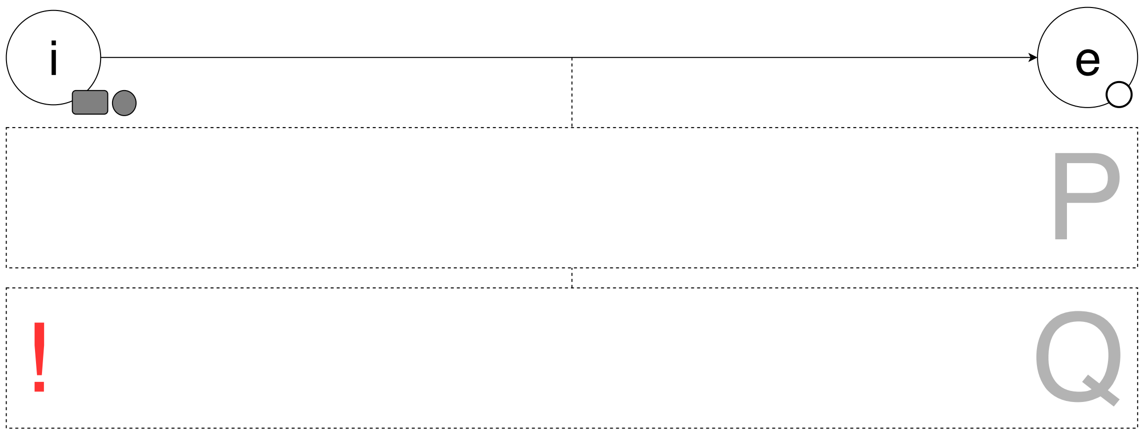


Figure 3.11.: DMQL pattern for nested conjunction.

The logical or is slightly more complex. At first, there is no means to express that m out of n subpatterns of a path edge are required. However, we can use De Morgan's law to express the logical or using the logical and. Consider the following example formula:

$$EF(\neg EFp \vee EFq) \equiv EF(\neg(EFp \wedge \neg EFq)) \tag{3.8}$$

Due to the outer negation, the problem changed into expressing that a path does not have certain properties, i.e. subpatterns. The solution is to forbid the path to feature itself at a whole where the subpatterns are fulfilled. Nested positive operators such as EFq map to forbidden subpatterns while negated operators like $\neg EFp$ map to required subpatterns this time, due to the transformation. The respective DMQL query is shown in figure 3.12. It features the basic path $i \rightarrow e$ but also includes a nested path $i' \rightarrow e$ (subpattern 1). The goal is to make both refer to the same model path. If they do, the nested path is mapped

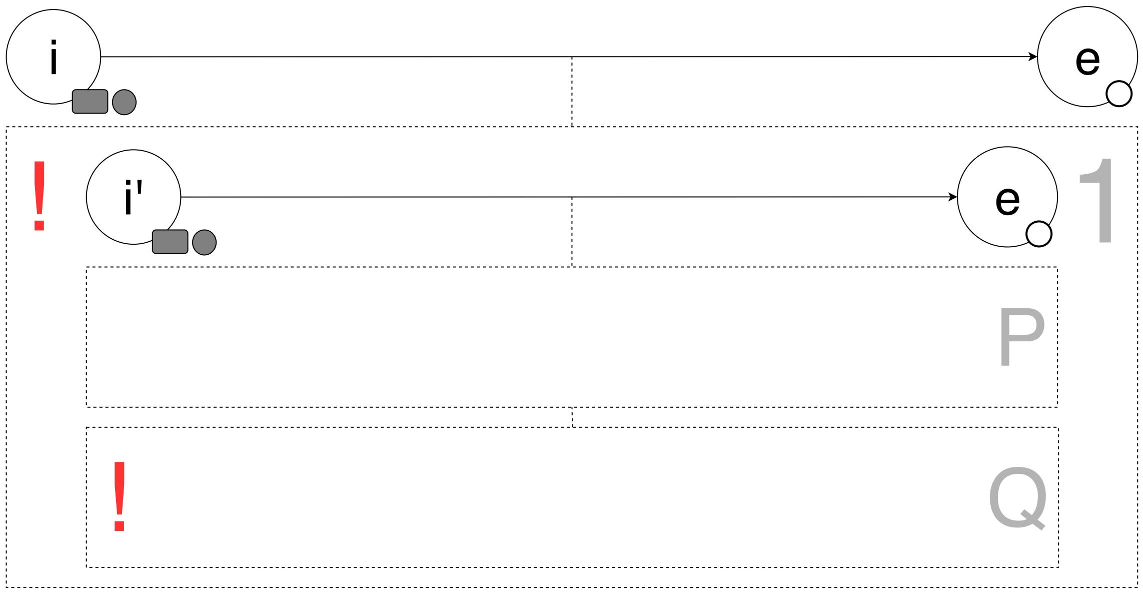


Figure 3.12.: DMQL pattern for nested disjunction and nested logical implication.

if the parental path features all subpatterns defined for the nested path, i.e. P is required and Q is forbidden. In this case the nested path is mapped successfully and invalidates the parental path as it is forbidden to lie on it. If the path is not mapped successfully, the root path stays valid.

This pattern can not stand alone. While the two pattern vertices e automatically refer to the same model vertex (there is exactly one end event per path, as defined in section 3.3.2), i and i' may refer to different vertices. However, if the paths should refer to the same path, i and i' must refer to the same vertex, too. Thus, we use a global rule to enforce that the pattern vertices i and i' do refer to the same model vertex. For example, we instrument unique captions or enrich model vertices with unique identifiers beforehand. In our example the rule could look as follows: $i.id = i'.id$. Defined global rules are checked before evaluating the path intersection. There may be multiple paths (starting at the model vertex i has been mapped to) that end in an end event vertex and thus map to subpattern 1. However, exactly one of them lies on the parental path completely: the parental path itself. Hence, only the parental path can invalidate itself by featuring all specified subpatterns. In result, this construct will match the parental path only if it either fulfills Q or does not fulfill P. Using global rules to tie pattern vertices and let them map to the same model vertex is a key mechanism for nesting and combining nested operators in the form of subpatterns. For subsequent queries we assume such global rules to be existent whenever we use the same vertex identifier within a DMQL patten and its subpatterns. Specifically, we use the same vertex identifier i for all pattern vertices $i_1, i_2, ..., i_n$ to imply that there is a global rule of the form $i_1.id = i_2.id = ... = i_n.id$.

Note that we just modeled the formula

$$EF(\neg EFp \wedge EFq) \equiv EF(AF\neg p \wedge EFq) \tag{3.9}$$

which is mixing AF and EF in a single query. Pattern lookups are the root level are always existential but forbidden subpatterns actually can proof the absence of certain patterns regarding the matched model path. In fact, the previous example already constitutes the logical implication. Logical equivalence can be expressed by combining the presented subpattern mechanisms, analogously to combining the mapping results of root operators in the previous section. Figure 3.13 visualizes a query that reconstructs $EFp \Leftrightarrow EFq$ where the subpatterns P and Q model EFp and EFq as before.

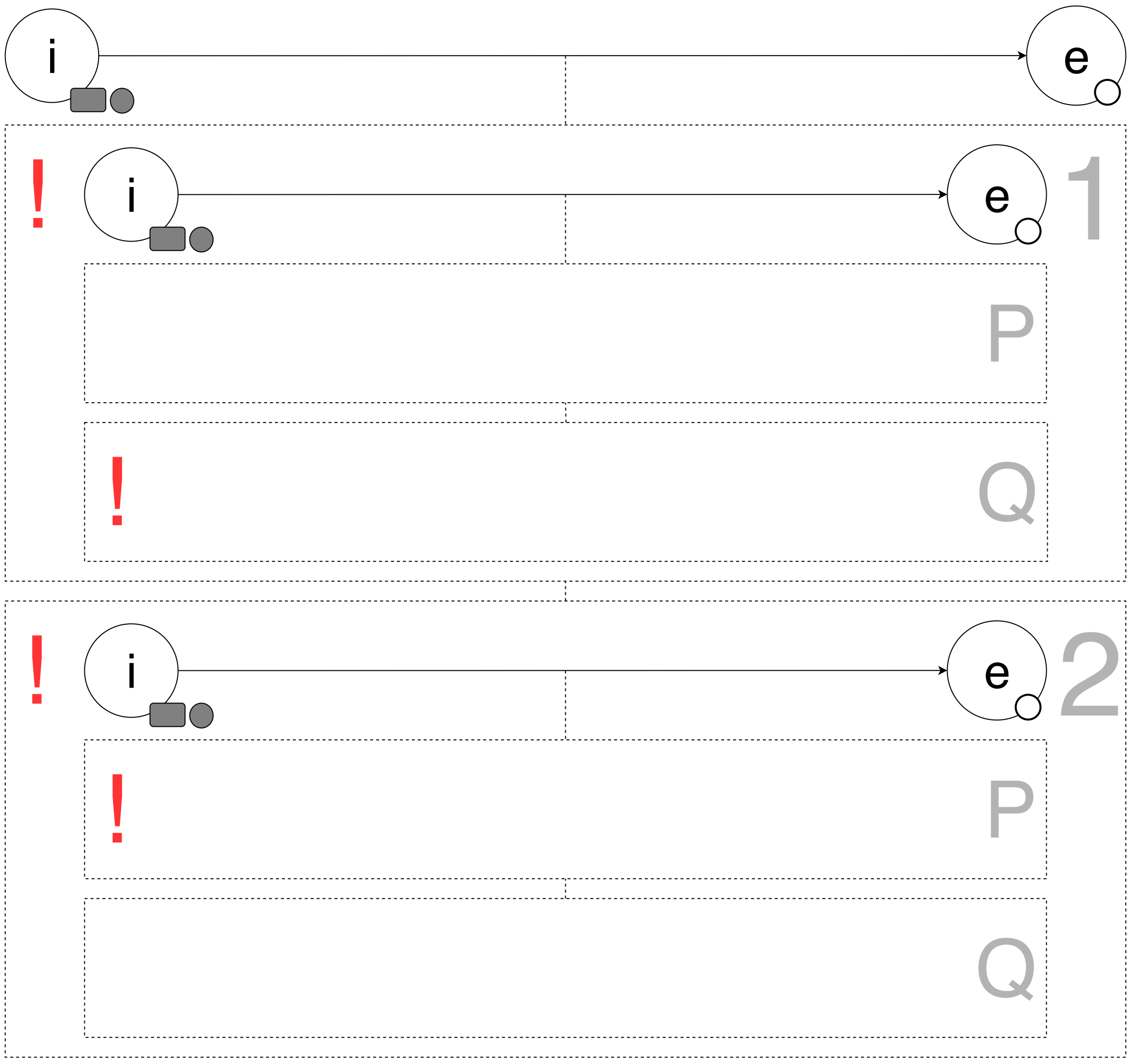


Figure 3.13.: DMQL pattern for nested logical equality.

3.3.5. Chaining CTL Operators

Considering the BNFs for state and path formulas (see section 2.2.2.1), CTL operators can be nested into each other, such as in $EF(EGp)$. Nesting refers to chaining formulas into each other by using the target state of one formula as the start state of a second formula. Given two DMQL patterns that represent the formulas, this translates to merging the patterns by starting the second pattern with the target vertex of the first one. Each formula can be a nested operator but also a logical combination of nested operators. According to the

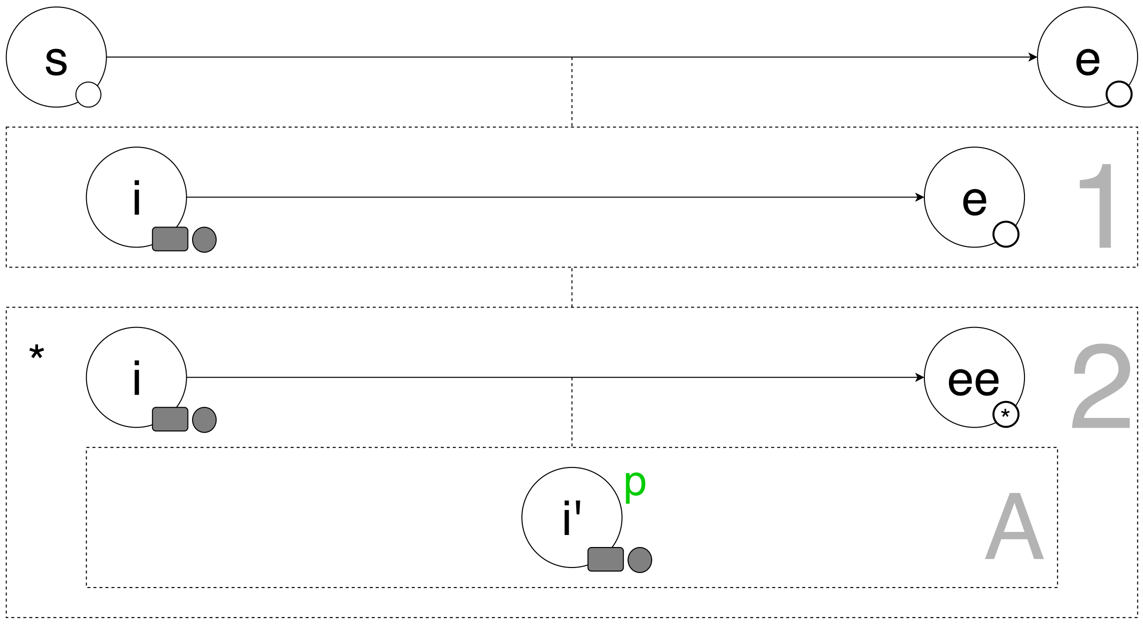


Figure 3.14.: Preliminary DMQL pattern representing the CTL formula $EF(EFp)$.

mechanisms described in section 3.3.4, we can reconstruct any logical combination of nested operators if we manage to express the operators as subpatterns. In previous sections, all operators were defined as patterns with the main logic inside subpatterns. This time, however, each operator has to be defined as a subpattern alone. The present section shows how the CTL operators can be expressed as subpatterns for existential queries. Expressing subpatterns for universal queries is analogous and left to the reader.

3.3.5.1. EF

Reconsider the DMQL pattern reconstructing EFp, where p was a proposition (figure 3.7). Pattern vertex i matches to model vertices that fulfill the EF operator, namely because p holds. When nesting a formula into EF, i has to match to model vertices that fulfill the nested formula instead. Consider the following CTL formula as an example, where EFp is nested in EF:

$$EF(EFp) \tag{3.10}$$

The respective DMQL pattern is presented in figure 3.14. The root path and subpattern 1 are largely copied from the original EF pattern and jointly model the outer EF. Proposition p has been removed from i. Instead, i must fulfill the nested formula EFp, i.e. there has to be a path starting in i which fulfills EFp. Though it is allowed, it is not required that the path which is fulfilling EFp lies on the parental path (s to e) completely. Thus, making the respective subpattern required (*req*) would be inappropriate. The subpattern needs to be partially required (*reqp*) because a single common element, namely i, is sufficient for the two paths. However, using *reqp* does not automatically imply that the common elements include i. Thus, we apply global rules again (see section 3.3.4): We demand that the pattern vertices i from subpattern 1 and subpattern 2 refer to the same model vertex. Again, these rules are assumed to be existent whenever vertices of a pattern,

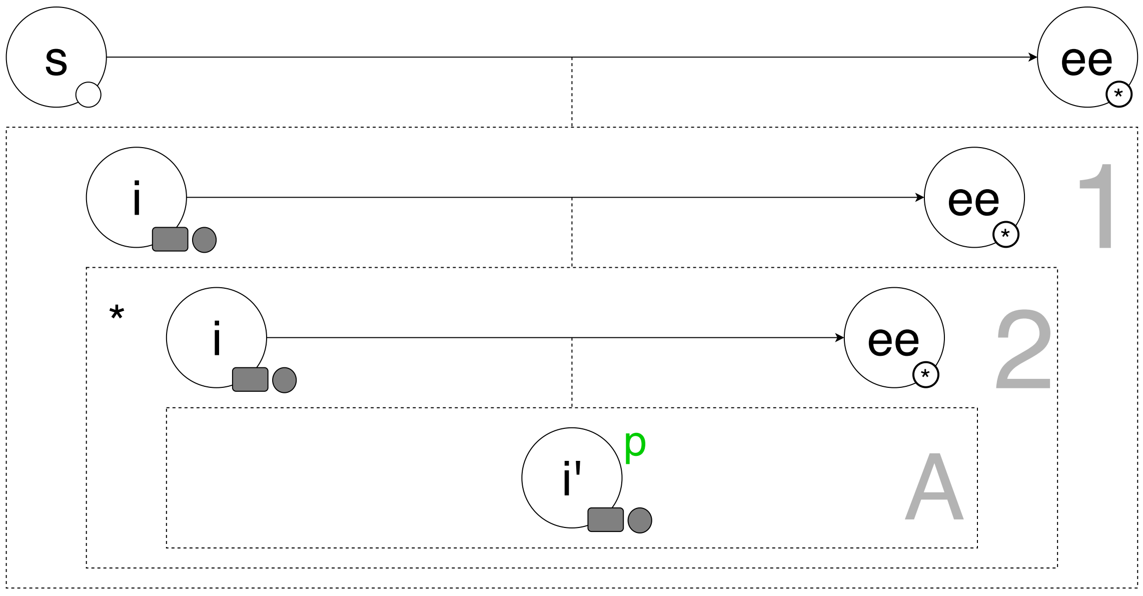


Figure 3.15.: DMQL pattern representing the CTL formula $EF(EFp)$.

including its subpatterns, use the same identifier. The global rule filters potential matches for subpattern 2 which start at the model vertex that i (subpattern 1) has been mapped to.

Left open is how the subpattern for the nested EFp (subpattern 2) exactly looks like. The semantic of EFp requires that any reachable state vertex fulfills p, starting from i. One could use i as the start vertex s of the nested EF pattern and end up with a path from i to e. However, this path would only be existent if $i \neq e$ which is not necessarily true. The solution is to add an artificial end event ee that all end event vertices are connected to. The path from i to ee will always exist if we imply $i \neq ee$. Finally, the second path must contain the target vertex i' of the nested EF (subpattern 2A), in which proposition p holds. Note that the implication $i \neq ee$ is actually desired now, as we do not want any proposition or formula to be fulfilled by the artificial vertex. This is enforced by using a distinct vertex type for ee (note the modified event shape) so that it is not even considered as a state vertex and can not be matched to i', for example. Incorporating ee allows to express the nesting using a single subpattern, as shown in figure 3.15, which allows arbitrary nesting. The example described how EF can be nested into other patterns and how patterns can be nested into EF, simultaneously. Note that the root path and subpattern 1 model the outer EF while subpattern 2 models the nested EF. In fact, every operator pattern can be nested into EF by replacing subpattern 2 and EF can be nested into every operator pattern by adding subpattern 2 appropriately.

3.3.5.2. EX

Similarly to EF, reconsider the DMQL pattern for EXp (figure 3.3). When nesting a formula into EX, i has to match to model vertices that fulfill the nested formula. Consider

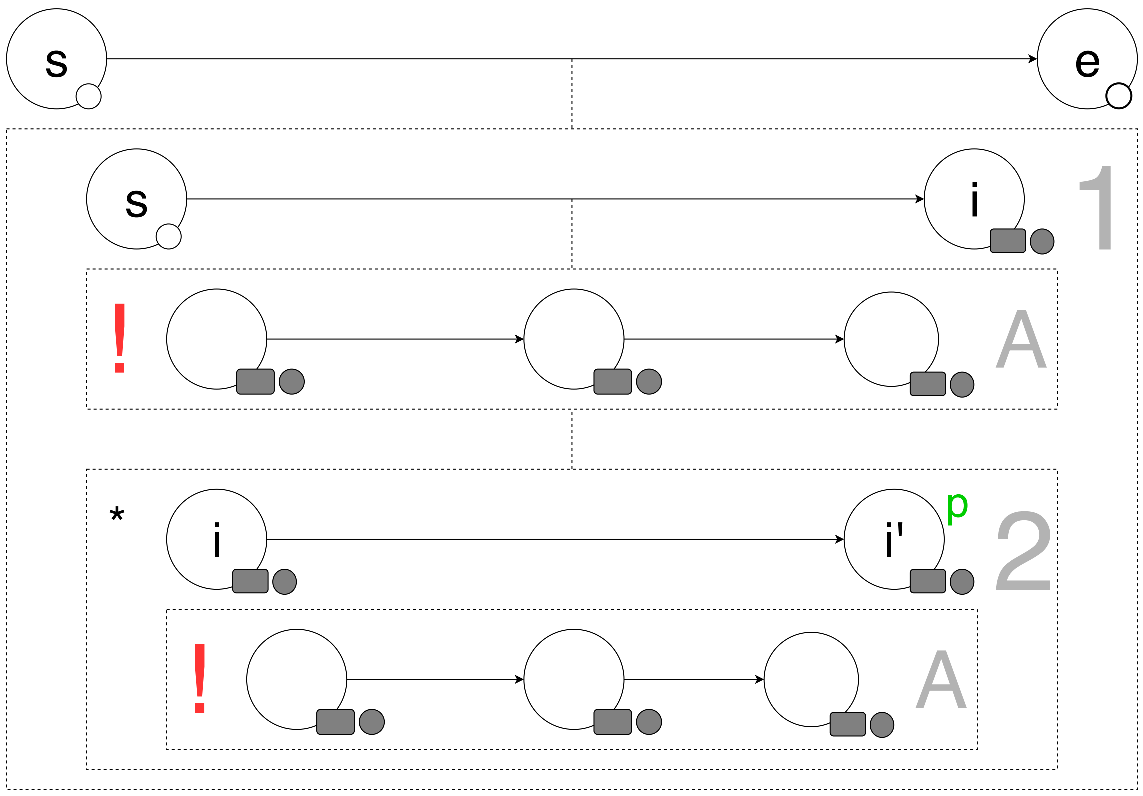


Figure 3.16.: DMQL pattern representing the CTL formula $EX(EXp)$.

the following CTL formula where EXp is nested in EX:

$$EX(EXp) \tag{3.11}$$

The respective DMQL pattern is shown in figure 3.16. The root path and subpattern 1 are copied from the original EX pattern but proposition p has been removed from i again. Instead, i must fulfill the nested EXp, i.e. there has to be a path starting in i which fulfills EXp. Analogously to EF, we reuse the subpattern of the original pattern but let it start from i and make it partially required (see subpattern 2). Global rules ensure that the path of subpattern 2 starts at i (from subpattern 1) while $reqp$ is relaxed enough to consider all potential matches. Subpattern 2A was left untouched and guarantees that i' is consecutive to i, similarly to subpattern 1A. Note that there was no need for the artificial vertex ee this time: If the path is not existent (i.e. $i = e$), EXp can not be fulfilled. While this example shows how to nest EX into EX, subpattern 2 models the nested EX and can be replaced appropriately to nest other operators into EX. Analogously, the subpattern can be added to other patterns to nest EX into the respective formula.

3.3.5.3. EG

Reconsider the DMQL pattern for EGp, as shown in figure 3.9. To show how EG can be nested into EG consider the following CTL formula.

$$EG(EGp) \tag{3.12}$$

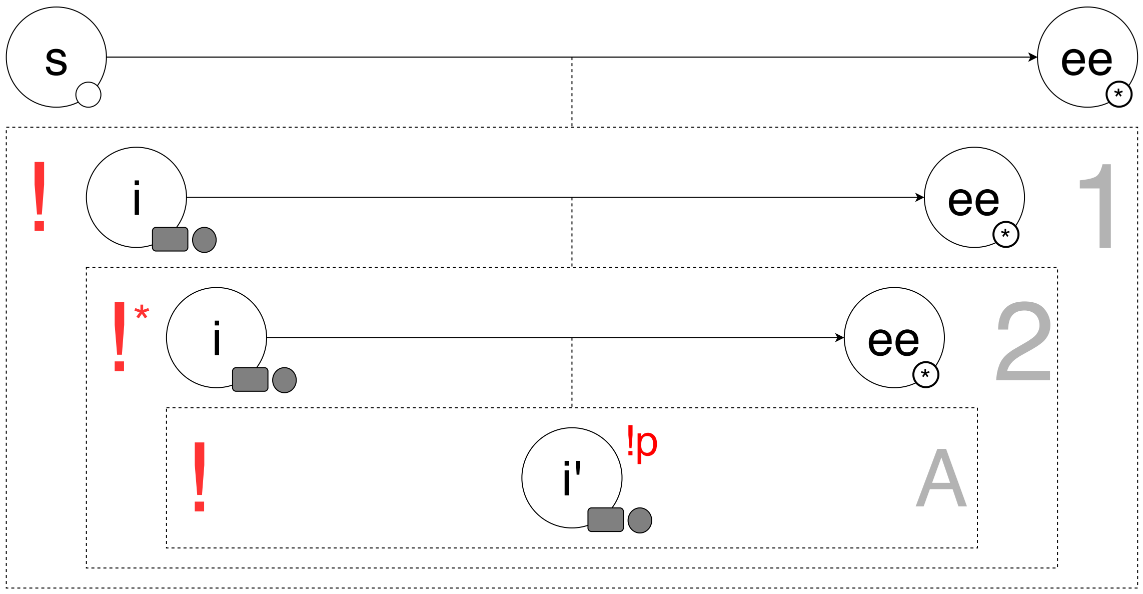


Figure 3.17.: DMQL pattern representing the CTL formula $EG(EGp)$.

Note that subpattern 1 of the original pattern guarantees that the root path is free from a state vertex i in which EG is violated, i.e. in which p does not hold. Similarly to EF, this subpattern does not feature a path. However, a path is required to nest subpatterns that model nested formulas. Therefore we deploy the artificial end event vertex ee again, as shown in figure 3.17: Subpattern 1 still guarantees that the root path is free of a state vertex i in which the outer EG is violated. The outer EG is violated if $\neg EGp$ holds in any state, analogously to $\neg p$ in the original pattern. $\neg EGp$ holds if there is no path starting at i (subpattern 2) that fulfills the nested EGp. Note that EG thus requires the nested formula subpattern to be $forb$ instead of $reqp$. EGp, in turn, holds if the path is free from state vertices in which p does not hold (subpattern 2A). In result, subpattern 1 models $\neg EF(\neg EGp) \equiv EG(EGp)$ which is the desired result. Note that subpattern 2, including its necessity modifier, models

$$\neg EF\neg EF\neg p \equiv \neg\neg EF\neg p \equiv EF\neg p \equiv \neg EGp \equiv AG\neg p \tag{3.13}$$

Thus we just showed how to nest universal operator patterns into existential formula patterns. As stated in section 3.3.4 this is possible because $forb$ can actually show the absence of certain patterns with regard to the matched model path. This example shows how EG can be nested into EG. In fact, EG can also be nested into patterns other than EG by adding subpattern 2 appropriately. For example, when nesting EGp into EF, subpattern 2 of figure 3.14 would be replaced by subpattern 2 of the described pattern. Note that the subpattern necessity is not replaced, i.e. subpattern 2 would be partially required when copied to the EF pattern. This is because the outer pattern determines the required subpattern necessity, depending on whether the nested path should be existent ($reqp$) or not ($forb$), i.e. whether the represented formula should hold or not. As stated, this switch allows to express existential and universal operators in existential queries. Similarly, subpattern 2 of figure 3.17 can be replaced appropriately to nest other operators into EG.

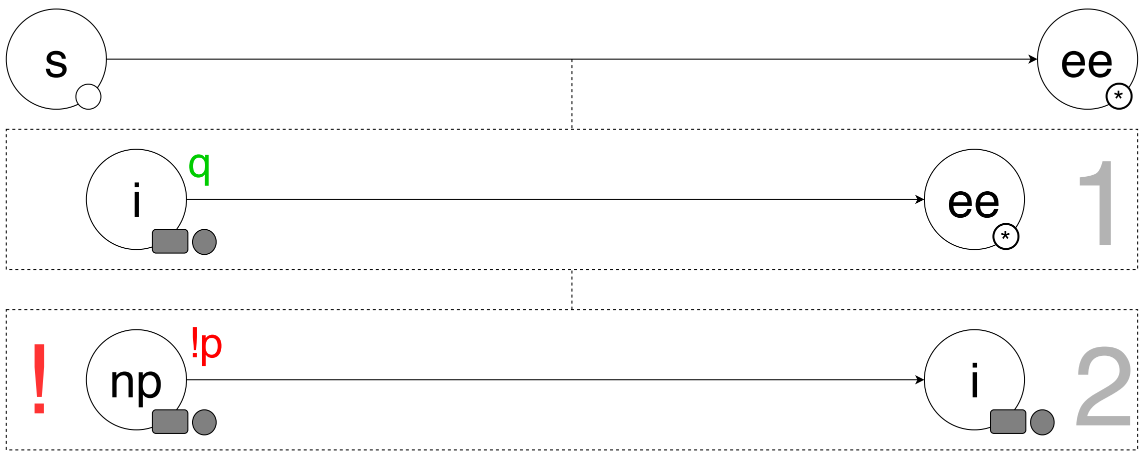


Figure 3.18.: Improved version of the DMQL pattern reconstructing $pEUq$.

3.3.5.4. EU

Before presenting how the pattern for EU can be nested, we identify the potential to simplify and improve the original operator pattern for $pEUq$ (figure 3.5) using recently introduced mechanisms, namely the artificial end event and global rules. As described in section 3.3.3.3, the operator pattern firstly implies $i \neq s$ and thus did not work when proposition q was holding in the start vertex s. Previously, we introduced the artifical end event vertex ee to change the implication $i \neq e$ to $i \neq ee$. In the improved pattern, we change subpattern 1 to a path from i to ee (as shown in figure 3.18) to resolve any implication issues, i.e. i can now be matched to s and e. Secondly, subpattern 1A of the original pattern ensured that no intermediate state vertex violates $pEUq$, i.e. that p holds in every intermediate state vertex. Intermediate, in this context, means on the path starting from s but ending in the vertex before i. As such, p must hold in successor of i but not in i itself. In the improved pattern, we remove this subpattern and add an interlinked subpattern 2 instead: On the path from s to i, no successor of i is allowed to violate $pEUq$, i.e. $\neg p$ does not hold in any succeeding state vertex. Using this subpattern requires global rules to link the two pattern vertices i, a mechanism which was not used in the original pattern. The resulting pattern includes both predicates once and thus leads to a reduced effort for nesting.

In contrast to previous operators, $pEUq$ has two arguments which may be nested formulas instead of propositions. Consider the following CTL formula as an example

$$(EFa \wedge EFb)EU(pEUq) \tag{3.14}$$

where (1) q has been replaced by $pEUq$ to show how EU is nested (2) and p has been replaced by a non-atomic formula to utilize the mechanisms for nested logical combination and negation, presented in section 3.3.4. Starting with the improved EU pattern, we first remove q from i in subpattern 1, thus i is only required to be a state vertex on the root path. Since we replace q with $pEUq$, i now has to fulfill this formula. Therefore we add the improved pattern as a nested subpattern, starting in i instead of s, as shown in subpattern 1A of figure 3.19. The subpattern is partially required to match any path that starts in i. Subpattern 2 originally models the absence of a successor np of i that violates proposition p. Since p has been replaced with $EFa \wedge EFb$, subpattern 2 internally needs to model that at least one of the EF operators does not hold in np. Formally, this equals to the disjunction $\neg EFa \vee \neg EFb$. As described in section 3.3.4, a disjunction is not possible in DMQL directly but needs to be modeled via negation and conjunction: We model that the path np to i in subpattern 2 is not a path (subpattern 2A) where both EFa (subpattern 2Aa) and EFb (subpattern 2Ab) hold. Subpattern 2 (without its necessity modifier $forb$) matches to paths from np to i. Subpattern 2A models exactly the same path but is forbidden, thus subpattern 2 will only match a path if subpattern 2A, at the same time, does not. Subpattern 2A (without its necessity modifier $forb$) matches paths from np to i where both EF operators hold in np. As a result, the path in subpattern 2 will only match if its np does not model both EFa and EFb. Since subpattern 2 is forbidden, the respective root path and the matched i are invalidated if such an np exists. In combination, subpattern 1 and subpattern 2 model the example formula. This more complex example shows how logically combined EF operators and the EU operator can be nested into EU. In fact, EU can be nested into other patterns by adding subpattern 1A appropriately and every operator pattern can be nested into EU. Note, however, that two subpatterns (namely subpattern 1A and subpattern 2A) model the nesting, as EU takes two arguments.

3.4. Mapping DMQL to CTL

After mapping the CTL operators to DMQL queries, DMQL features are now mapped to CTL formulas. Again, we need to stress out the different semantics of DMQL and CTL. While DMQL queries retrieve sets of model elements that allow the specified mapping, CTL formulas are evaluated to a binary result. Therefore we are interested in mapping $M(Q) = \varnothing \Leftrightarrow E(f) = false$, where E is an evaluating function for CTL formulas. DMQL queries are not evaluated in initial vertices only, thus retrieved paths may start at any vertex. Since CTL formulas are evaluated in initial states, they generally need to be wrapped by the EF operator to simulate the semantics of DMQL queries.

As described in section 2.3.2.2, DMQL pattern graphs (i.e. queries) consist of three types of components. Section 3.4.1 describes how vertices and their properties are mapped by CTL. Edges and their properties are handled in section 3.4.2. Finally, section 3.4.3 shows the mapping of global rules to CTL features.

3.4.1. Pattern Vertices and their Properties

This section presents CTL formulas that reconstruct DMQL patterns consisting of vertices and their properties. Pattern vertices can feature three built-in properties, namely *vid*, *vcaption* and *vtypes* (see section 2.3.2.2). The following subsections show how atomic propositions are used in combination with EF to reproduce patterns involving a vertex

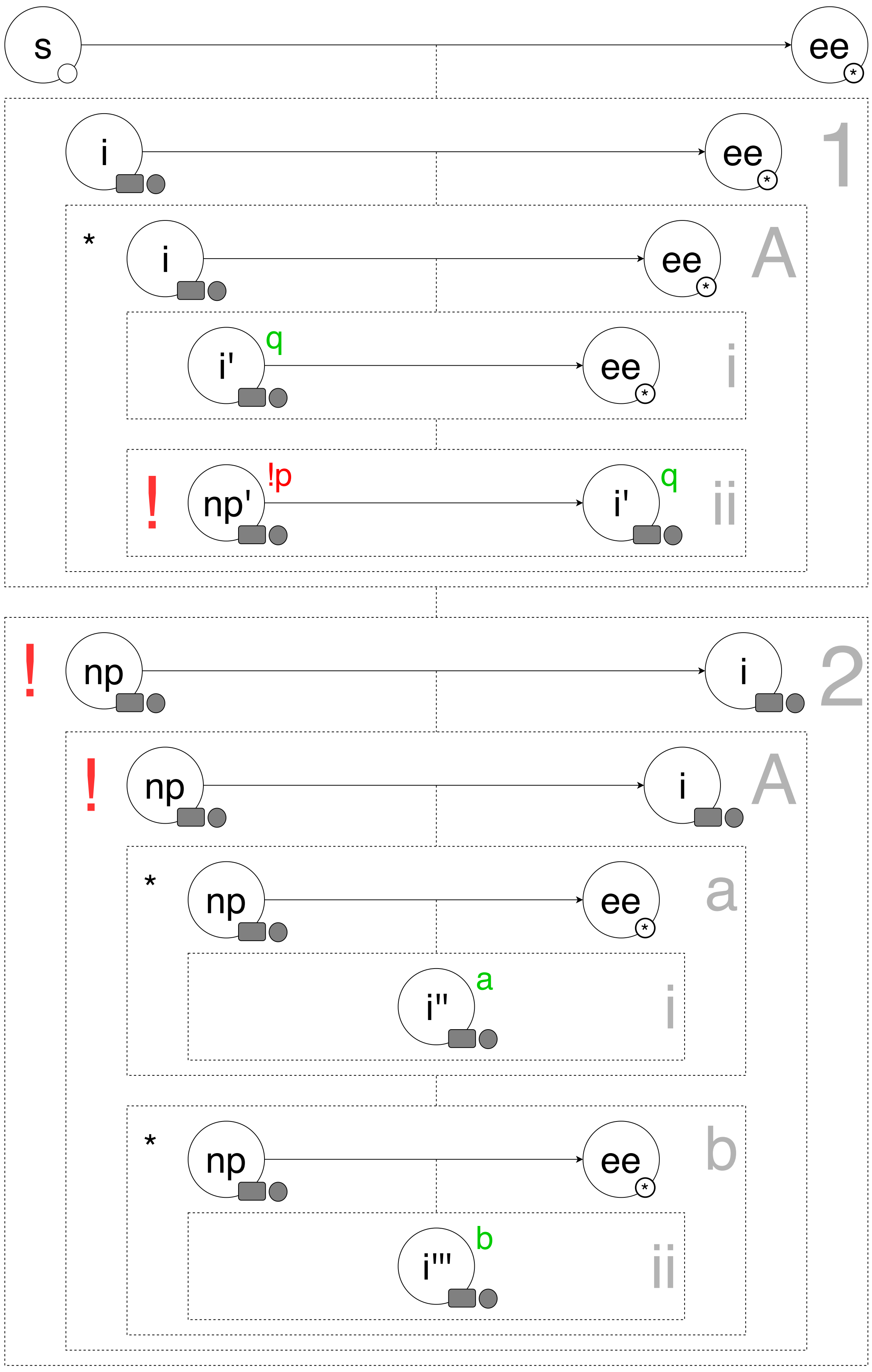
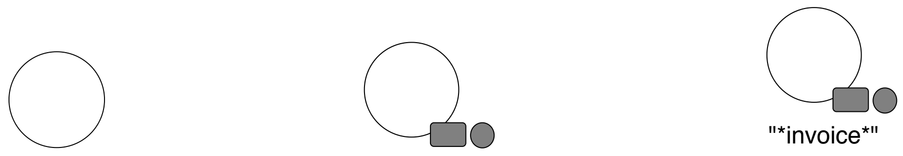


Figure 3.19.: DMQL pattern representing the CTL formula $(EFa \wedge EFb)EU(pEUq)$.



(a) match any vertex (b) with certain types (c) with various properties

Figure 3.20.: DMQL patterns incorporating vertices and their properties.

and its properties. Note that DMQL patterns can not incorporate multiple vertices, if they are not connected by edges.

3.4.1.1. Plain Vertex

Consider a DMQL pattern that matches any vertex in a process model graph (figure 3.20a). Its *vid* is unset or unused, $vcaption = *$ matches any vertex and $vtypes = \varnothing$ indicates that any vertex type is appropriate for the mapping. The following CTL formula reconstructs this pattern:

$$EF(true) \tag{3.15}$$

EF moves the allowed start state from initial states to any states in the system. There are no restrictions for the vertex, thus *true* is used as expression which evaluates to true in any state.

3.4.1.2. The vid Property

The *vid* property has only a single purpose: Providing an identifier to refer to the particular vertex from global rules. Thus the property itself does not have to mapped by CTL but possible global rules do (see section 3.4.3).

3.4.1.3. The vcaption Property

The property *vcaption* limits the mapping of the respective pattern vertex to model vertices having captions which match the specified expression. Assuming the vertex caption is encoded as an atomic proposition *caption* in states, CTL can incorporate it:

$$EF(caption = \text{"}MyCaption\text{"}) \tag{3.16}$$

The state is simply required to feature the desired caption.

3.4.1.4. The vtypes Property

The vertex property *vtypes* allows to define a set of allowed vertex types. For example, figure 3.20b matches any state vertex, i.e. $vtypes = \{activity, event\}$. Assuming the vertex type is encoded as an atomic proposition *type*, the following CTL formula mimics the presented pattern.

$$EF(type = activity \vee type = event) \tag{3.17}$$

The state is required to feature any of the allowed type values.

3.4.1.5. Combining Properties

Previous patterns made use of single vertex properties. Figure 3.20c combines multiple vertex properties: It matches state vertices which are related to invoices, regarding their caption. The following CTL formula reconstructs the described pattern.

$$EF((type = activity \vee type = event) \wedge caption = "*invoice*") \tag{3.18}$$

3.4.2. Pattern Edges and their Properties

This section presents CTL formulas that reconstruct DMQL patterns with edges and their properties. Pattern edges need a start and an end vertex, thus these queries always involves named two vertices. See section 3.4.1 on how pattern vertices and their properties map to CTL. Given two vertices, however, edges can be established. Pattern edges can feature 14 properties in total (see section 2.3.2.2). The following subsections show how edges and their properties map to CTL. First, plain simple and path edges are mapped. However, plain edges are not very useful and just for the sake of completeness. Subsequent subsections present the mappings of the numerous edge properties.

3.4.2.1. Plain Simple Edge

A simple edge from one vertex to another (as shown in figure 3.1d earlier) implies that there are at least two vertices where there is a path with minimum and maximum length 1 from one to the other. The following CTL formula maps this DMQL pattern:

$$EF(EX(true)) \tag{3.19}$$

EF is used to specify the start state as an arbitrary state (see section 3.4.1) while $EX(true)$ ensures the existence of a second state. Chaining the parts together in the shown way implies that the second state is reachable from the first within one step.

3.4.2.2. Plain Path Edge

In contrast to simple edges, a path edge between to vertices matches subgraphs where there are at least two vertices on a path of arbitrary maximum length from the first to the second. A DMQL pattern of this kind was shown in figure 3.1b and the following CTL formula reconstructs this behavior:

$$EF(EX(EF(true))) \tag{3.20}$$

This formula is very similar to the previous formula for simple edges. Again, EF relaxes the start state and $EX(EF(true))$ ensures the existence of a second state. This time, however, the second EF relaxes the constraint and solely implies that the second state is reachable from the first - using any number of steps which is at least 1, due to the EX.

3.4.2.3. The eid Property

Analogously to the *vid* property of vertices, the edge property *eid* does not need to be mapped itself. Instead, the mapping of global rules is presented in section 3.4.3, which make use of this property.

3.4.2.4. The ecaption Property

Analogously to the *vcaption* property of vertices, the *ecaption* property can be mimicked by CTL if the captions are encoded as atomic propositions. Since atomic propositions are only available for states, edges need to be encoded as system states if the *ecaption* property is required. While this is perfectly possible, it makes the formulation of CTL formulas more cumbersome. Thus presented formulas will assume that no edge states are present, as long as they are not required.

3.4.2.5. The dir Property

The edge property $dir \subseteq P(\{org, opp, none\})$ describes which direction an edge can be interpreted in (P denotes the power set function). Our default is $dir = \{org\}$ and the direction of the (directed) model edge has to match the direction modeled in the pattern. In CTL, this direction is given by the structure of the formula - more precisely, by the order in which the operators, jointly reconstructing the DMQL pattern elements, are chained. Reconsider the simple path edge where the respective CTL formula $EF(EX(EF(true)))$ was mapping the default direction. This formula is equivalent to $EF(true \wedge EX(EF(true))$ where the first $true$ is a "restriction" for the start state. Now let's use $p = true$ and $q = true$ for the two states, to allow for a better understanding of the next step:

$$EF(p \wedge EX(EF(q)) \tag{3.21}$$

This formula still equals $EF(EX(EF(true)))$ and thus handles cases where $dir = \{org\}$. If the direction would have been opp instead, the order of the CTL formula had to be reversed:

$$EF(q \wedge EX(EF(p))) \tag{3.22}$$

Now the end vertex state is looked up first and the start vertex state has to be reachable from it. If the direction would have been $\{opp, opp\}$, the CTL formulas had to be combined into a disjunction:

$$EF(p \land EX(EF(q)) \lor EF(q \land EX(EF(p))) \tag{3.23}$$

In fact, the direction used in figure 3.1d is *none*. Transitions between system states always have a direction and can not model undirected edges directly. However, for an edge type which is known to be undirected, $dir = \{none\}$ equals $dir = \{org, opp\}$. As stated, to query properties that are encoded in edges, such as the direction, edges have to be encoded as system states. Henceforth, previous formulas are only valid for (directed) graphs encoded in a system without edge states. Adapting them to such systems is trivial but is left out for the sake of readability (see above). The DMQL pattern visualized in figure 3.1d, however, uses an undirected path edge and thus requires a CTL formula which is aware of vertex and edge states in the system. The following formula reconstructs this pattern:

$$EF(p \land EX(isUndirected \land EX(EF(q)))) \lor EF(q \land EX(isUndirected \land EX(EF(p)))) \tag{3.24}$$

Note that the formula assumes that p and q only hold for states modeling vertices and $isUndirected$ only holds for states modeling edges.

3.4.2.6. Length Properties

Two edge properties steer the minimum and maximum length of the path, namely minl and maxl. Addressing paths, these properties do not require edge states in the system. CTL can use the EX operator to reconstruct the $minl$ property. This operator effectively pushes a condition forward to subsequent states on validated traces. In fact, the plain path edge formula already used EX to model $minl = 1$. EX may be nested arbitrarily to model the desired $minl$ while a final EF keeps the maximum length relaxed. For example, the following CTL formula reconstructs a path edge with $minl = 3$, similar to the pattern in figure 3.1c.

$$EF(p \land EX(EX(EX(EF(q))))) \tag{3.25}$$

If the final EF is left out, only the state with the specified distance can fulfill the formula, thus $minl = maxl$. However, the maximum length can also be variable, i.e. $maxl > minl$. The easiest way to reconstruct this using CTL is to form a disjunction of allowed lengths, as shown for $maxl = 3$ in the following formula:

$$EF(p \land EX(q) \lor EX(EX(q)) \lor EX(EX(EX(q)))) \tag{3.26}$$

This method can also be applied if both $minl$ and $maxl$ are incorporated in a pattern. The following formula reconstructs a pattern with a single path edge from p to q where $minl = 2$ and $maxl = 4$:

$$EF(p \land EX(EX(q)) \lor EX(EX(EX(q))) \lor EX(EX(EX(EX(q))))) \tag{3.27}$$

In this case, expressions violating the minimum length of the desired path are removed from the disjunction produced in the previous step.

3.4.2.7. Vertex and Edge Type Properties

The edge properties *vtypesr* and *vtypesf* steer which vertex types are allowed or forbidden on the path. Given that vertex types are encoded as atomic propositions in states, these can be incorporated into CTL formulas. Consider a DMQL pattern where a vertex type, e.g. *event*, is forbidden on a path from p to q. In CTL, this pattern could be expressed as:

$$EF(p \land EX(EF(E(type \neq event)\, U q))) \tag{3.28}$$

Though this formula does not feature the exact semantic of the pattern (i.e. the formula stops at the first occurrence of q) it is adequate in the context of the restricted mapping from DMQL to CTL. While it is possible to limit a path to a certain vertex type, it is not possible to make a certain vertex type required. This is because we can not keep track of which vertex types already occurred on the current trace. Analogously, the edge properties *etypesr* and *etypesf* restrict paths according to occurred edge types. Similarly to *vtypesf*, *etypesf* can be mimicked using the *EU* operator:

$$EF(p \land EX(EF(E(etype \neq myEdge)\, U q))) \tag{3.29}$$

3.4.2.8. Vertex and Edge Overlap Count Properties

The properties *minvo* and *maxvo* allow to specify the minimum and maximum number of vertices that have to overlap on the model path in order to be mapped to the pattern path. Basically, this requires to count how often certain elements have been visited or how often they occur in the current trace. This count then has to be within the defined interval. However, CTL lacks of a dedicated construct to count states. Furthermore, the language CTL does not feature constructs to address the current trace, with their start and end states. Traces are solely used internally for the evaluation of formulas. This limitations applies to states in general, independent of whether they model vertices or edges. Thus, the related edge properties *mineo* and *maxeo* can not be adequately represented in CTL, too. A prominent use case for the overlap properties is the detection of loops (Delfmann, Breuker, Matzner & Becker 2015). Loops can be detected, if a loop's vertex is known in advance and identifiable, e.g. incorporating an identifier. The simple path edge formula is already able to detect this loop, assuming that $p = q$ and that p only holds in the targeted state, i.e. is the identifier. However, there is no means in CTL to express that an arbitrary or each state must (not) occur a given number of times, due to the lack of means to count or remember trace elements other than using the formula structure.

3.4.2.9. The Subpattern Property

The subpattern property of path edges limits matching model paths to those that feature either (i) all (ii) at least one (iii) not all (iv) none of the elements that the subpattern has been matched to. As described, DMQL paths map to traces in CTL, thus this asks for the intersection of traces. In fact, all CTL operators do ask for a specific intersection: They require that valid traces start at the current state. Effectively, this asks for an intersection between the current trace and evaluated traces where the evaluated traces start in the current state on the current trace. However, the four types of subpattern necessity require a different semantic. For example, *reqp* (at least one element required) requires that any trace element overlaps with the trace - and not just the first one. There is no language construct in CTL to express such arbitrary intersections of traces and it can not be mapped to the available intersection semantic.

Obviously, some DMQL patterns using subpatterns can be reconstructed, such as those reconstructing the CTL operators. However, all of them either model the semantics of a CTL operator intentionally or can be expressed without subpatterns. For example, the EU DMQL pattern presented in figure 3.18 exactly models the semantics of the EU operator: There is a path (or trace) starting in the current vertex (or state) leading to a vertex (state) where q eventually holds (subpattern 1) and where p holds until it has been reached (subpattern 2). While the EU operator stops at the first occurrence of q, DMQL queries would consider all matching paths but this is irrelevant in the context of our restricted mapping. The DMQL pattern for EF (figure 3.7) also contains a subpattern. While it does model the semantics of the EF operator, the subpattern is not strictly required and could be modeled without. The DMQL pattern for EX (figure 3.3) uses subpatterns to skip gateways in the model graph while still limiting valid vertices to subsequent next state vertices. In CTL, we assumed that gateways were not modeled in the system and thus do not need to be skipped. If we apply this assumption to the model graph, the EX pattern shrinks to a simple edge and would not include a subpattern either. The DMQL pattern for EG (figure 3.9 uses a subpattern to check for the absence of a certain predicate on the path. This mimics the EG operator which can also be expressed as $EG\,p \equiv E\neg F\neg p$. Again, this is exactly what the subpattern expresses, what it has been designed for.

While these patterns were designed to model the exact logic of constructs available in CTL (i.e. its operators), arbitrary patterns can not be reconstructed using CTL. It has no means to express intersections other than the described start state intersection.

3.4.3. Global Rules

Global rules filter matched subgraphs according to the properties of their elements, possibly pattern-crossing. An example are the frequently deployed global rules to interlink pattern vertices. They ensure that the model vertices, which pattern vertices from different patterns have been mapped to, feature the same identifier. As defined in section 2.3.2.1, there are four properties of graph elements that may be used in global rules: (1) The element caption,

(2) the type of the vertex and (3) the number of incoming, outgoing or undirected edges connected to the vertex. However, we assume that every vertex property would be possible, not only captions and types.

Some of these properties are structural properties, such as the number of incoming, outgoing or undirected edges, which require counting. As CTL lacks of constructs for counting, these can not be mapped. Regular vertex properties can be mapped by CTL if their context can be mapped, too. For example, consider two subpatterns where certain vertices should feature the same identity. CTL inherently maps these rules if the subpattern try to fix the start point of certain paths (e.g. subpattern 2Aa and 2Ab in figure 3.19 for $\boldsymbol{EFa} \wedge \boldsymbol{EFb}$) or if they fix the end point of certain paths (e.g. $\boldsymbol{EF}(a \wedge b)$). Other global rules are mapped because a CTL operator happens to model the same semantic, e.g. in case of $\boldsymbol{EU}$ where subpattern 1 and subpattern 2 (figure 3.19 again) refer to the same vertices. However, CTL can only maps these identity rules. For example, it can not express that two states have the same proposition value, e.g. the state fulfilling $\boldsymbol{EFa}$ should have the same value for $\boldsymbol{type}$ as the state fulfilling $\boldsymbol{EFb}$. Naturally, if a subpattern is already not expressible in CTL, its global rules consequently can not be mapped to CTL neither.

4. Summary and Discussion

This chapter aims to summarize and discuss the previous chapter where CTL and DMQL have been mapped to each other as far as possible. The two languages have not been mapped to each other exactly but with limitations and assumptions. Nevertheless, the extent of the achieved partial mappings (i.e. given the aforementioned limitations) differs. To best describe the extent of the partial mappings an overview is given and related limitations are discussed later on, to make meaningful statements about the advantages of either language type, according to the representative language.

Table 4.1 provides an overview about the reconstruction of CTL features, using DMQL and vice versa. All basic CTL operators have been successfully mapped to single DMQL patterns, given that the operator parameters are propositions (e.g. EFp). The queries for these operator patterns can be combined to model the logical combination of any type and number of operators (e.g. $EFp \land \neg EFq$). The operator patterns can be nested in arbitrary depth, using subpatterns that are interlinked by global rules, to model nested operators (e.g. $EX(EXp)$). Nested operator patterns can be logically combined using negation, conjunction and interlinked subpatterns to model complex formulas such as $EF(AGp \lor EXq)$. Explicitly, this allows to nest universal operator patterns into existential queries and vice versa. However, the discussion following this overview qualifies the scenarios in which the described extent of mapping is factually valid. In contrast, CTL can not reconstruct all features of DMQL patterns. Patterns with a vertex and its properties can be reconstructed by mapping vertices to states and properties to the state's atomic propositions. Simple edges can be mapped using EX while plain path edges (i.e. with arbitrary length but without additional properties) can be mapped using EF. CTL is able to map certain yet not all edge properties. Basic properties such as captions or the edge direction can be expressed by using atomic propositions respectively reversing the formula structure. Length restrictions can be mapped via combinations of the operators EX and EF at the cost of duplicating parts of the formula. Forbidden vertex and edge types can be excluded from traces by CTL but it can not track whether all required types have occurred.

	CTL				DMQL															
	Operators				Vertices			Edges: Plain		Edges: Properties							Global Rules		
Features	Standalone	Combined	Nested	Both	Plain	vtypes	vcaption	Plain Edge	Plain Path	ecaption	dir	*typesr	*typesf	Lengths	Overlaps	Subpattern	*id	caption	Edge Counts
DMQL	✓	✓	✓	✓															
CTL					✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	i	✗	✗

Table 4.1.: Overview of CTL reconstruction using DMQL.

Mapping path overlap properties requires to count the overlapping path elements. CTL provides no mechanisms for counting, instead it has to be involved in formulas. While this allows to count how many times a specific state occurred (given an identifying expression), counting the occurrences of arbitrary states is not possible. Forbidden subpatterns can be reconstructed via the ***EU*** operator as long as they are atomic. However, there are no means to express non-atomic subpatterns or subpatterns with different necessity modifiers. Exceptions are subpatterns which happen to model the semantics of a CTL operator or a logical combination of them. In particular, required subpatterns can not be mapped as CTL is not aware of traces [1]. Global rules are largely unmapped by CTL, especially those referring to structural properties (e.g. the node degree). Since global rules basically interlink subpatterns (which are largely unmapped), being able to map global rules would not increase the mapping extent significantly. Though CTL can reconstruct some DMQL concepts the mapping underlies the same restrictions as the reverse mapping. This is because DMQL works on structural graphs and CTL works on behavioral system models. Another limitation for the mapping from DMQL to CTL is time: The first mapping took more than half of the time we invested on mapping and thus will be more progressed.

The presented overview visualizes the extent to which CTL and DMQL have been able to reconstruct each other's features. Due to the differences between paths and traces, these features do not map exactly. Exemplary, we determine in which scenarios the paths in DMQL patterns factually map to all traces the reconstructed CTL formula would have considered as valid. These paths are directed paths from start to end events. As a result, matched model paths solely consist of control flow elements (i.e. are free from data objects etc.), represent a sequence of executed models elements and thus model traces. BPMN (section 2.1.1) defines three categories of control flow elements: activities, events and gateways. Depending on involved control flow elements, the directed paths stop modeling (all) valid traces: Activities and events typically are executed in the specified order, i.e. successors are enabled as soon as the predecessor has been executed. This property is used by the directed paths to express that an activity follows another. In fact, this rudimentary property does not even apply to all activities. Some activities are not embedded in the control flow, such as ad-hoc activities. Including these activities would go beyond the scope of this thesis but their exclusion shows up a limitation. Given a model consisting of embedded activities and events only, the model execution is linear: Every start event is connected to a single end event, there is only one path and it maps to the only, valid trace.

Gateways change this linearity and redefine when an activity follows another. This is because gateways have conditional branches (*xor*), spawn parallel flow in additional branches (*and*) or both (*or*). Subsequent activities in conditional branches might be executed if the respective conditions hold. Activities on different parallel branches are executed in parallel, i.e. may precede or succeed each other even if they are not connected by a directed path. Consider a BPMN model with a branching ***and*** gateway that spawns a second branch, as shown in appendix D.1. The activities ***A*** and ***B*** are executed in parallel, hence any order

[1]Traces are artifacts for the evaluation of formulas, used within model checkers.

is acceptable. As a result, there are two valid traces: One where A is executed first and one where A is executed last. Note that none of the traces maps to a directed path from s to e. Consequently, presented patterns would not detect these execution sequences. Given more time we could develop patterns that cover parallel flow, for now figure D.2 shows how the presented patterns can be modified: State vertices follow each other if (1) they are connected by a directed path (as modeled before) or (2) both reside in different branches of a parallel gateway, i.e. the paths from the gateway to the activities do not include the respective other activity. Instead of using modified patterns, the existing patterns could be applied to transformations of the process model graphs as well. For example, every valid trace can be mapped to a directed path in the Kripke structure which CTL has been defined on. However, producing transformations with this property would probably inherit problems related to model checking, e.g. the state space explosion problem.

Gateways of the type *xor* and *or* are conditional, i.e. the activation of their branches depends on the fulfillment of specified conditions. These conditions may depend on the execution of prior elements. DMQL does not evaluate the conditions but implicitly assumes that they are evaluated to both true and false for at least once. Given that a certain execution sequence prohibits a condition to become true or false any longer, DMQL retrieves paths which do not map to valid traces. For example, consider the simplified BPMN model visualized in figure 4.1: Activity ***EnterPassword*** (denoted as a) must be executed at maximum three times to activate the branch with activity ***ShowDashboard*** (denoted as b), as specified by the conditional edges of the *xor* gateway. The directed path in the presented

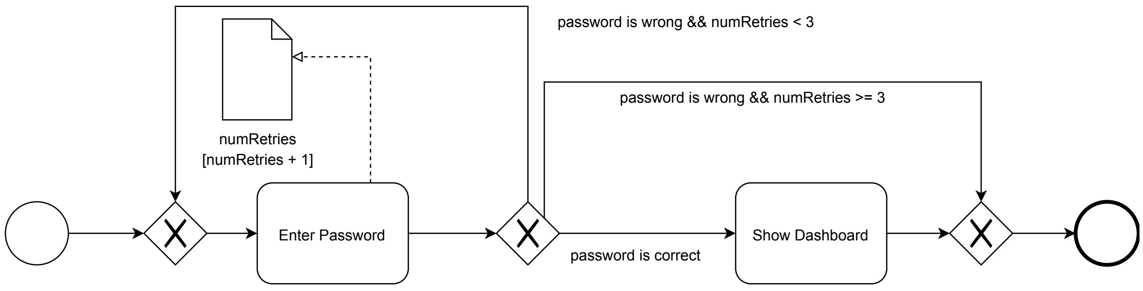


Figure 4.1.: Example BPMN model with interdependent conditional flow.

EF pattern (figure 3.7) would fetch any path from a to b (see subpattern 1) independently from how many times a has been repeated on that path. Of course, some invalid paths could be excluded on pattern level manually (see figure D.3) but this would quickly yield complex patterns which are cumbersome to create, read and maintain. For an automatic solution DMQL could evaluate conditions before traversing conditional edges. However, it has been stated that the semi-formal BPMN and other modeling languages are lacking of exact execution semantics (c.f. (Polyvyanyy et al. 2017)) and difficulties could arise when achieving such an evaluation. It is questionable whether and how CTL would achieve the evaluation of valid traces if the language or the produced model do not define sufficient execution semantics, especially for the variables used in conditions. In case CTL does not, its traces would neglect conditional as well and all directed paths would continue to model traces.

5. Conclusion

The mapping of CTL and DMQL, representing temporal logic and graph pattern matching respectively, has been achieved with different extents and limitations. We presented DMQL patterns that reconstruct every CTL formula, given the absence of (1) elements which are not embedded in the control-flow, (2) parallel gateways and (3) conditional gateways. The first type of elements is out of scope. Parallel gateways cause parallel flow which the presented patterns do not handle. We described how the patterns can be modified to cope with parallel flow. Conditional gateways make the activation of branches dependent on conditions. DMQL neglects these conditions and retrieves paths which are no valid execution sequences, in return. Depending on whether CTL traces respect these conditions, DMQL patterns model different semantics than the CTL formulas they aim to represent. In this case the retrieved false positives can be filtered out on the pattern level manually. Pattern matching might even be extended to evaluate the conditions of edges before their traversal to avoid retrieving false positives in the first place. Otherwise, patterns achieve the same semantics as the reconstructed formulas. CTL can reconstruct some DMQL features but largely fails to map structural queries and subpatterns, as CTL lacks of dedicated counting mechanisms and is not aware of the traces its evaluated on. To conclude, CTL explores the state space to find valid traces (i.e. process execution sequences) while DMQL retrieves paths that represent structurally possible executions, neglecting conditional control flow. As a result, CTL is more suitable for behavioral and DMQL is more suitable for structural querying. While CTL seems inappropriate for structural querying, DMQL can be used for behavioral querying given that there are no false positives (with respect to CTL) or that their existence is acceptable. These advantages and disadvantages finally answer our initial research question.

The contributions of this thesis are as follows: We presented DMQL patterns for all CTL operators and how they can be combined to model complex formulas. Analogously, we presented CTL formulas that reconstruct certain features of DMQL patterns. We discussed the extent of these mappings and pointed out significant differences between traces and paths, depending on involved control flow elements. While these differences qualify the mapping extent, we described how DMQL patterns can be modified to eliminate them partially.

Mapping CTL to DMQL and vice versa was exemplary to assess the advantages of temporal logic and graph pattern matching. Arguably, other languages (such as LTL, VMQL, etc.) exist and may show different mapping results. Though both partial mappings have been presented, we could not spend the same resources on both and as a result, the second mapping (DMQL to CTL) was not researched as extensively as the first. Furthermore, the presented mappings are theoretical. They have not been tested at large scale and runtime performance benchmarks are not provided. To assess the mapping between temporal logic and graph pattern matching in greater detail, future work could further investigate the influence of BP model elements on the mapping of traces and paths. This influence, however, may depend on the particular modeling language. Besides modeling new patterns, automatic mechanisms can be investigated where transformations of the model graph are used to answer behavioral queries in DMQL. For example, Kripke structures are graphs where each path is a valid trace. However, generating these structures may involve the state space generation and inherit its problems. In this context, the mappings should be tested and benchmarked to evaluate their applicability in the real world. Finally, instead of trying to achieve a general mapping between these conceptually different types of languages, it may be more promising to map subsets of their features, adopted to certain use cases such as compliance checking.

Bibliography

Andrews, T. et al. (2003), 'Business Process Execution Language for Web Services (BPEL), Version 1.1', `http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf`. Accessed: 2019-09-30.

Awad, A. (2007), BPMN-Q: A Language to Query Business Processes, in 'EMISA', Vol. 119, pp. 115–128.

Awad, A., Decker, G. & Weske, M. (2008), Efficient Compliance Checking Using BPMN-Q and Temporal Logic, in 'International Conference on Business Process Management', Springer, pp. 326–341.

Awad, A., Polyvyanyy, A. & Weske, M. (2008), Semantic Querying of Business Process Models, in '2008 12th International IEEE Enterprise Distributed Object Computing Conference', IEEE, pp. 85–94.

Awad, A. & Sakr, S. (2012), 'On Efficient Processing of BPMN-Q Queries', Computers in Industry 63(9), 867–881.

Awad, A., Weidlich, M. & Weske, M. (2011), 'Visually Specifying Compliance Rules and Explaining their Violations for Business Processes', Journal of Visual Languages & Computing 22(1), 30–55.

Becker, J., Bergener, P., Delfmann, P. & Weiß, B. (2011), Modeling and checking business process compliance rules in the financial sector, in 'Thirty Second International Conference on Information Systems'.

Beeri, C., Eyal, A., Kamenkovich, S. & Milo, T. (2006), Querying Business Processes, in 'Proceedings of the 32nd international conference on Very large data bases', VLDB Endowment, pp. 343–354.

Beeri, C., Eyal, A., Kamenkovich, S. & Milo, T. (2008), 'Querying Business Processes with BP-QL', Information Systems 33(6), 477–507.

Bräuer, S., Delfmann, P., Dietrich, H.-A. & Steinhorst, M. (2013), Using a Generic Model Query Approach to Allow for Process Model Compliance Checking – An Algorithmic Perspective, in 'Proceedings of the 11th International Conference on Wirtschaftsinformatik (WI)(2013). Leipzig, Germany', pp. 1245–1259.

Browne, M. C., Clarke, E. M. & Grümberg, O. (1988), 'Characterizing finite kripke structures in propositional temporal logic', Theoretical Computer Science 59(1-2), 115–131.

Clarke, E. M. & Emerson, E. A. (1981), Design and synthesis of synchronization skeletons using branching time temporal logic, in 'Workshop on Logic of Programs', Springer, pp. 52–71.

Clarke, E. M., Emerson, E. A. & Sistla, A. P. (1986), ‘Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications’, ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2), 244–263.

Clarke Jr, E. M., Grumberg, O., Kroening, D., Peled, D. & Veith, H. (2000), Model Checking, MIT press.

Cruz, E. F., Machado, R. J. & Santos, M. Y. (2012), From Business Process Modeling to Data Model: A systematic approach, in ‘2012 Eighth International Conference on the Quality of Information and Communications Technology’, IEEE, pp. 205–210.

De Nicola, A., Missikoff, M. & Smith, F. (2012), ‘Towards a method for business process and informal business rules compliance’, Journal of software: Evolution and Process 24(3), 341–360.

Delfmann, P., Breuker, D., Matzner, M. & Becker, J. (2015), ‘Supporting Information Systems Analysis Through Conceptual Model Query – The Diagramed Model Query Language (DMQL)’, Communications of the Association for Information Systems 37.

Delfmann, P., Herwig, S., Lis, Ł., Stein, A., Tent, K. & Becker, J. (2010), ‘Pattern Specification and Matching in Conceptual Models - A Generic Approach Based on Set Operations’, Enterprise Modelling and Information Systems Architectures (EMISAJ) 5(3), 24–43.

Delfmann, P., Steinhorst, M., Dietrich, H.-A. & Becker, J. (2015), ‘The generic model query language GMQL – Conceptual specification, implementation, and runtime evaluation’, Information Systems 47, 129–177.

El Kharbili, M., de Medeiros, A. K. A., Stein, S. & van der Aalst, W. M. (2008), ‘Business Process Compliance Checking: Current State and Future Challenges’, MobIS 141, 107–113.

Elgammal, A., Turetken, O., van den Heuvel, W.-J. & Papazoglou, M. (2016), ‘Formalizing and appling compliance patterns for business process compliance’, Software & Systems Modeling 15(1), 119–146.

Emerson, E. A. & Halpern, J. Y. (1986), ‘“Sometimes” and “Not Never” Revisited: On Branching versus Linear Time Temporal Logic’, Journal of the ACM (JACM) 33(1), 151–178.

Förster, A., Engels, G., Schattkowsky, T. & Van Der Straeten, R. (2007), Verification of Business Process Quality ConstraintsBased on Visual Process Patterns, in ‘First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE’07)’, IEEE, pp. 197–208.

Gori, M., Maggini, M. & Sarti, L. (2005), The RW2 Algorithm for Exact Graph Matching, in ‘International Conference on Pattern Recognition and Image Analysis’, Springer, pp. 81–88.

Group, O. M. (2014), 'Business Process Model and Notation (BPMN), Version 2', `https://www.omg.org/spec/BPMN/`. Accessed: 2019-08-26.

Group, O. M. (2019), 'About OMG', `https://www.omg.org/index.htm`. Accessed: 2019-08-26.

Hammer, M. & Champy, J. (1993), Reengineering the Corporation: A Manifesto for Business Revolution, Harper Business.

Kammerer, K., Kolb, J. & Reichert, M. (2015), PQL - A Descriptive Language for Querying, Abstracting and Changing Process Models, in 'Enterprise, Business-Process and Information Systems Modeling', Springer, pp. 135–150.

Liu, Y., Muller, S. & Xu, K. (2007), 'A static compliance-checking framework for business process models', IBM Systems Journal 46(2), 335–361.

Pesic, M. (2008), Constraint-based workflow management systems: shifting control to users, PhD thesis, Department of Industrial Engineering & Innovation Sciences.

Pesic, M., Schonenberg, H. & Van der Aalst, W. M. (2007), DECLARE: Full Support for Loosely-Structured Processes, in '11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)', IEEE, pp. 287–287.

Pesic, M. & Van der Aalst, W. M. (2006), A Declarative Approach for Flexible Business Processes Management, in 'International conference on business process management', Springer, pp. 169–180.

Pnueli, A. (1977), The temporal logic of programs, in 'Foundations of Computer Science, 1977., 18th Annual Symposium on', IEEE, pp. 46–57.

Polyvyanyy, A. (2018), Business Process Querying, in 'Encyclopedia of Big Data Technologies', Springer.

Polyvyanyy, A., Ouyang, C., Barros, A. & van der Aalst, W. M. (2017), 'Process Querying: Enabling Business Intelligence through Query-Based Process Analytics', Decision Support Systems 100, 41–56.

Reisig, W. & Rozenberg, G. (1998), Lectures on Petri Nets I: Basic Models - Advances in Petri Nets, Springer Science & Business Media.

Riehle, D. M. (2018), Checking Business Process Models for Compliance – Comparing Graph Matching and Temporal Logic, in 'International Conference on Business Process Management', Springer, pp. 403–415.

Sakr, S. & Al-Naymat, G. (2010), 'Graph indexing and querying: a review', International Journal of Web Information Systems 6(2), 101–120.

Sakr, S. & Awad, A. (2010), A Framework for Querying Graph-Based Business Process Models, in 'Proceedings of the 19th international conference on World wide web', ACM, pp. 1297–1300.

Shasha, D., Wang, J. T. & Giugno, R. (2002), Algorithmics and Applications of Tree and Graph Searching, in 'Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems', ACM, pp. 39–52.

Smith, F., Missikoff, M. & Proietti, M. (2012), Ontology-based Querying of Composite Services, in 'Business System Management and Engineering', Springer, pp. 159–180.

Storrle, H. (2009), VMQL: A Generic Visual Model Query Language, in '2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)', IEEE, pp. 199–206.

Störrle, H. & Acretoaie, V. (2013), Querying Business Process Models with VMQL, in 'Proceedings of the 5th ACM SIGCHI Annual International Workshop on Behaviour Modelling-Foundations and Applications', ACM, p. 4.

Ter Hofstede, A. H., Ouyang, C., La Rosa, M., Song, L., Wang, J. & Polyvyanyy, A. (2013), APQL: A Process-model Query Language, in 'Asia-Pacific Conference on Business Process Management', Springer, pp. 23–38.

Valmari, A. (1996), The State Explosion Problem, in 'Advanced Course on Petri Nets', Springer, pp. 429–528.

van der Aalst, W. M., De Beer, H. & van Dongen, B. F. (2005), Process Mining and Verification of Properties: An Approach based on Temporal Logic, in 'OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"', Springer, pp. 130–147.

Vardi, M. Y. (2001), Branching vs. Linear Time: Final Showdown, in 'International Conference on Tools and Algorithms for the Construction and Analysis of Systems', Springer, pp. 1–22.

Venkatraman, M. & Singh, M. P. (1999), 'Verifying Compliance with Commitment Protocols', Autonomous agents and multi-agent systems 2(3), 217–236.

Wang, J., Jin, T., Wong, R. K. & Wen, L. (2014), 'Querying business process model repositories', World Wide Web 17(3), 427–454.

Weske, M. (2012), Business Process Management - Concepts, Languages, Architectures, 2nd Edition, Springer.

Zuck, L. (1986), 'Past temporal logic', Weizmann Institute of Science .

A. BPMN Meta Model for DMQL

The mapping between the two reference languages will be shown with BPMN models in mind. In order to use DMQL for a particular modeling language, a suitable meta model has to be specified. Thus, we formalize a rudimentary meta model for BPMN:

$$\rho = \{\text{BPMN}\}$$
$$\text{BPMN} = (T_v\text{BPMN}, T_e\text{BPMN})$$
$$T_{vBPMN} = \{\text{event}, \text{activity}, \text{gateway}, \text{data}\}$$
$$T_{eBPMN} = \{\text{ev-ea}, \text{ev-ae}, \text{evc-eg}, \text{evc-ge}, \text{cf}, \text{cf-g}, \text{cfc-ag}, \text{cfc-ga}, \text{dr}, \text{dw}\}$$
$$\text{ev-ea} = (\text{event}, \text{activity}), \text{ev-ae} = (\text{activity}, \text{event}), \text{evc-eg} = (\text{event}, \text{gateway}),$$
$$\text{evc-ge} = (\text{gateway}, \text{event}), \text{cf} = (\text{activity}, \text{activity}), \text{cf-g} = (\text{gateway}, \text{gateway}),$$
$$\text{cfc-ag} = (\text{activity}, \text{gateway}), \text{cfc-ga} = (\text{gateway}, \text{activity}),$$
$$\text{dr} = (\text{data}, \text{activity}), \text{dw} = (\text{activity}, \text{data})$$

Please note that this meta model is rudimentary and does not claim to be complete. Unsupported artefacts include group (having no semantic meaning) and annotation (typically unstructured, textual description). Consequently, the connection object association is unsupported which is used to connect these artefacts. Lastly, swimlanes (represent organizations) are unsupported. However, the meta model covers flow objects, data objects, sequence and message flow, arguably the BPMN core elements which are most important for BP models.

B. DMQL Pattern Formalization

While all DMQL patterns were presented using a specialized visual notation (see section 3.3.1), a formalization of the patter expresses them in a built-in notation. As summarized in section 2.3.2.1, DMQL patterns are generally formalized as $Q = (V_Q, E_Q, P_V, P_E, \delta, \varepsilon, \rho, G)$. The following patterns will define the vertices and edges alongside with their properties. Due to the avoidance of global rules, $G = \varnothing$ holds globally. See appendix A for a formalization of the meta model ρ.

B.1. Formalization of the DMQL Pattern for EX

$$
\begin{aligned}
&Q_{EX} = (V_{EX}, E_{EX}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EX} = \{v_s, v_e\}, E_{EX} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{EX'}\}) \\
&\theta_{EX'} = (Q_{EX'}, req)
\end{aligned} \tag{B.1a}
$$

$$
\begin{aligned}
&Q_{EX'} = (V_{EX'}, E_{EX'}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EX'} = \{v_s, v_i\}, E_{EX'} = \{e\}, e = (v_s, v_i, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).p = true \\
&\varepsilon(e) = (e, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{NxtSt}\}) \\
&\theta_{NxtSt} = (Q_{NxtSt}, forbp)
\end{aligned} \tag{B.1b}
$$

$$
\begin{aligned}
&Q_{NxtSt} = (V_{NxtSt}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{NxtSt} = \{v_1, v_2, v_3\} \\
&\delta(v_1) = (v1, "*", \{activity, event\}), \delta(v_2) = (v2, "*", \{activity, event\}) \\
&\delta(v_3) = (v3, "*", \{activity, event\})
\end{aligned} \tag{B.1c}
$$

B.2. Formalization of the DMQL Pattern for AX

$$
\begin{aligned}
&Q_{AX} = (V_{AX}, E_{AX}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{AX} = \{v_s, v_e\}, E_{AX} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{AX'}\}) \\
&\theta_{AX'} = (Q_{AX'}, forbp)
\end{aligned} \tag{B.2a}
$$

$$
\begin{aligned}
&Q_{AX'} = (V_{AX'}, E_{AX'}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{AX'} = \{v_s, v_i\}, E_{AX'} = \{e\}, e = (v_s, v_i, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).p = true \\
&\varepsilon(e) = (e, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{NxtSt}\}) \\
&\theta_{NxtSt} = (Q_{NxtSt}, forbp)
\end{aligned} \tag{B.2b}
$$

$$
\begin{aligned}
&Q_{NxtSt} = (V_{NxtSt}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{NxtSt} = \{v_1, v_2, v_3\} \\
&\delta(v_1) = (v1, "*", \{activity, event\}), \delta(v_2) = (v2, "*", \{activity, event\}) \\
&\delta(v_3) = (v3, "*", \{activity, event\})
\end{aligned} \tag{B.2c}
$$

B.3. Formalization of the DMQL Pattern for EU

$$
\begin{aligned}
&Q_{EU} = (V_{EU}, E_{EU}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EU} = \{v_s, v_e\}, E_{EU} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{EU'}\}) \\
&\theta_{EU'} = (Q_{EU'}, req)
\end{aligned} \tag{B.3a}
$$

$$
\begin{aligned}
&Q_{EU'} = (V_{EU'}, E_{EU'}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EU'} = \{v_s, v_i\}, E_{EU'} = \{e\}, e = (v_s, v_i, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).q = true \\
&\varepsilon(e) = (e, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{\bar{p}}\}) \\
&\theta_{\bar{p}} = (Q_{\bar{p}}, forbp)
\end{aligned} \tag{B.3b}
$$

$$
\begin{aligned}
&Q_{\bar{p}} = (V_{\bar{p}}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{\bar{p}} = \{v_p\} \\
&\delta(v_p) = (vp, "*", \{activity, event\}), \delta(v_p).p = false \wedge \delta(v_p).q = false
\end{aligned} \tag{B.3c}
$$

B.4. Formalization of the DMQL Pattern for AU

$$
\begin{aligned}
&Q_{AU} = (V_{AU}, E_{AU}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{AU} = \{v_s, v_e\}, E_{AU} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{AU'}\}) \\
&\theta_{AU'} = (Q_{AU'}, req)
\end{aligned} \tag{B.4a}
$$

$$
\begin{aligned}
&Q_{AU'} = (V_{AU'}, E_{AU'}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{AU'} = \{v_s, v_i\}, E_{AU'} = \{e\}, e = (v_s, v_i, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).q = true \\
&\varepsilon(e) = (e, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{\bar{p}}\}) \\
&\theta_{\bar{p}} = (Q_{\bar{p}}, forbp)
\end{aligned} \tag{B.4b}
$$

$$
\begin{aligned}
&Q_{\bar{p}} = (V_{\bar{p}}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{\bar{p}} = \{v_p\} \\
&\delta(v_p) = (vp, "*", \{activity, event\}), \delta(v_p).p = false \wedge \delta(v_p).q = false
\end{aligned} \tag{B.4c}
$$

B.5. Formalization of the DMQL Pattern for EF

$$
\begin{aligned}
&Q_{EF} = (V_{EF}, E_{EF}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EF} = \{v_s, v_e\}, E_{EF} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{EF'}\}) \\
&\theta_{EF'} = (Q_{EF'}, req)
\end{aligned}
\tag{B.5a}
$$

$$
\begin{aligned}
&Q_{EF'} = (V_{EF'}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EF'} = \{v_i\} \\
&\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).p = true
\end{aligned}
\tag{B.5b}
$$

B.6. Formalization of the DMQL Pattern for AF

$$
\begin{aligned}
&Q_{AF} = (V_{AF}, E_{AF}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{AF} = \{v_s, v_e\}, E_{AF} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{AF'}\}) \\
&\theta_{AF'} = (Q_{AF'}, forbp)
\end{aligned}
\tag{B.6a}
$$

$$
\begin{aligned}
&Q_{AF'} = (V_{AF'}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{AF'} = \{v_i\} \\
&\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).p = true
\end{aligned}
\tag{B.6b}
$$

B.7. Formalization of the DMQL Pattern for EG

$$
\begin{aligned}
&Q_{EG} = (V_{EG}, E_{EG}, P_V, P_E, \delta, \varepsilon, \rho, G) \\
&V_{EG} = \{v_s, v_e\}, E_{EG} = \{e_f\}, e_f = (v_s, v_e, 1) \\
&\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start \\
&\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end \\
&\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{EG'}\}) \\
&\theta_{EG'} = (Q_{EG'}, forbp)
\end{aligned}
\tag{B.7a}
$$

$$Q_{EG'} = (V_{EG'}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \tag{B.7b}$$
$$V_{EG'} = \{v_i\}$$
$$\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).p = false$$

B.8. Formalization of the DMQL Pattern for AG

$$Q_{AG} = (V_{AG}, E_{AG}, P_V, P_E, \delta, \varepsilon, \rho, G) \tag{B.8a}$$
$$V_{AG} = \{v_s, v_e\}, E_{AG} = \{e_f\}, e_f = (v_s, v_e, 1)$$
$$\delta(v_s) = (vs, "*", \{event\}), \delta(v_s).evPosType = start$$
$$\delta(v_e) = (ve, "*", \{event\}), \delta(v_e).evPosType = end$$
$$\varepsilon(e_f) = (ef, "*", \{org\}, 1, 0, 0, 0, 0, 0, \varnothing, \varnothing, \varnothing, \varnothing, \{\theta_{AG'}\})$$
$$\theta_{AG'} = (Q_{AG'}, req)$$

$$Q_{AG'} = (V_{AG'}, \varnothing, P_V, P_E, \delta, \varepsilon, \rho, G) \tag{B.8b}$$
$$V_{AG'} = \{v_i\}$$
$$\delta(v_i) = (vi, "*", \{activity, event\}), \delta(v_i).p = false$$

C. Negative Examples of DMQL Patterns

While the DMQL patterns presented in the thesis led to the desired results, this appendix section lists patterns that did not provide the intended semantics. The main purpose is the visualization of inappropriate concepts that a thesis section refers to and shall support the reader in following the written argumentation.

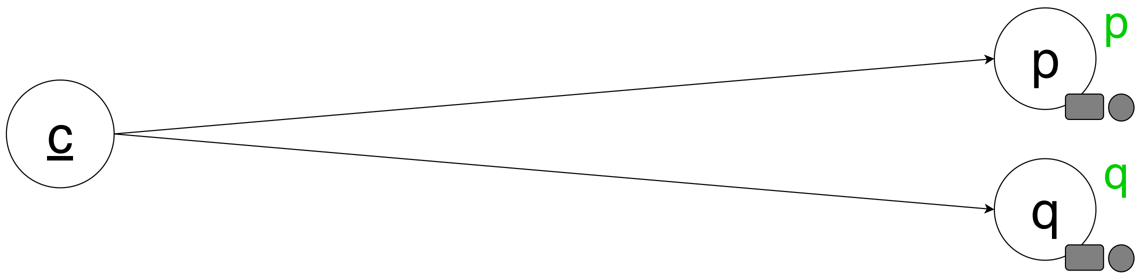


Figure C.1.: Inappropriate conjunction of DMQL paths.

D. Parallel and Conditional Control Flow

The models and patterns in this appendix are related to the discussion about parallel and conditional control flow in BP models which qualify the validity of DMQL patterns presented in this thesis (see chapter 4).

D.1. BPMN Model with Parallel Flow

The following simple BPMN model features parallel flow, changing the semantics of "subsequent control flow elements", such as activities: The strict need for a directed path from *A* to *B* has to be dropped as residing in different parallel branches is also sufficient for subsequent execution.

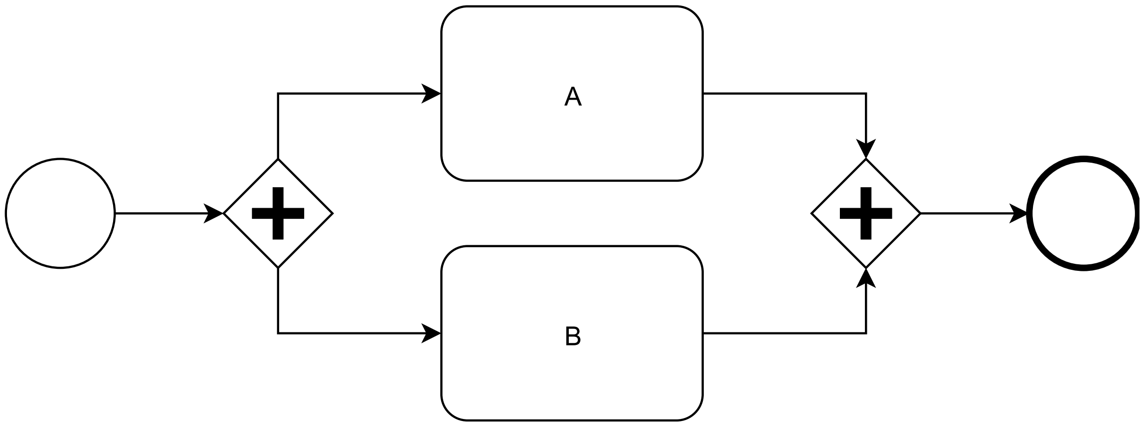


Figure D.1.: Example BPMN model with parallel flow.

D.2. DMQL Pattern Respecting Parallel Flow

The following DMQL pattern is a modification of the nested *EF* operator pattern (see section 3.3.5) capturing "b follows a" where a desired state vertex *b* (1) has to follow a vertex *a* directly (subpattern i) to capture sequential flow or (2) is on a different branch of a parallel gateway *g* than vertex *a*, to capture parallel flow. In the latter case, *g* is a common predecessor and *b* is not on the path from *g* to *a*. Note that if *a* was on the path from *g* to *b*, the first condition would have been met already, we do not need to handle this situation twice.

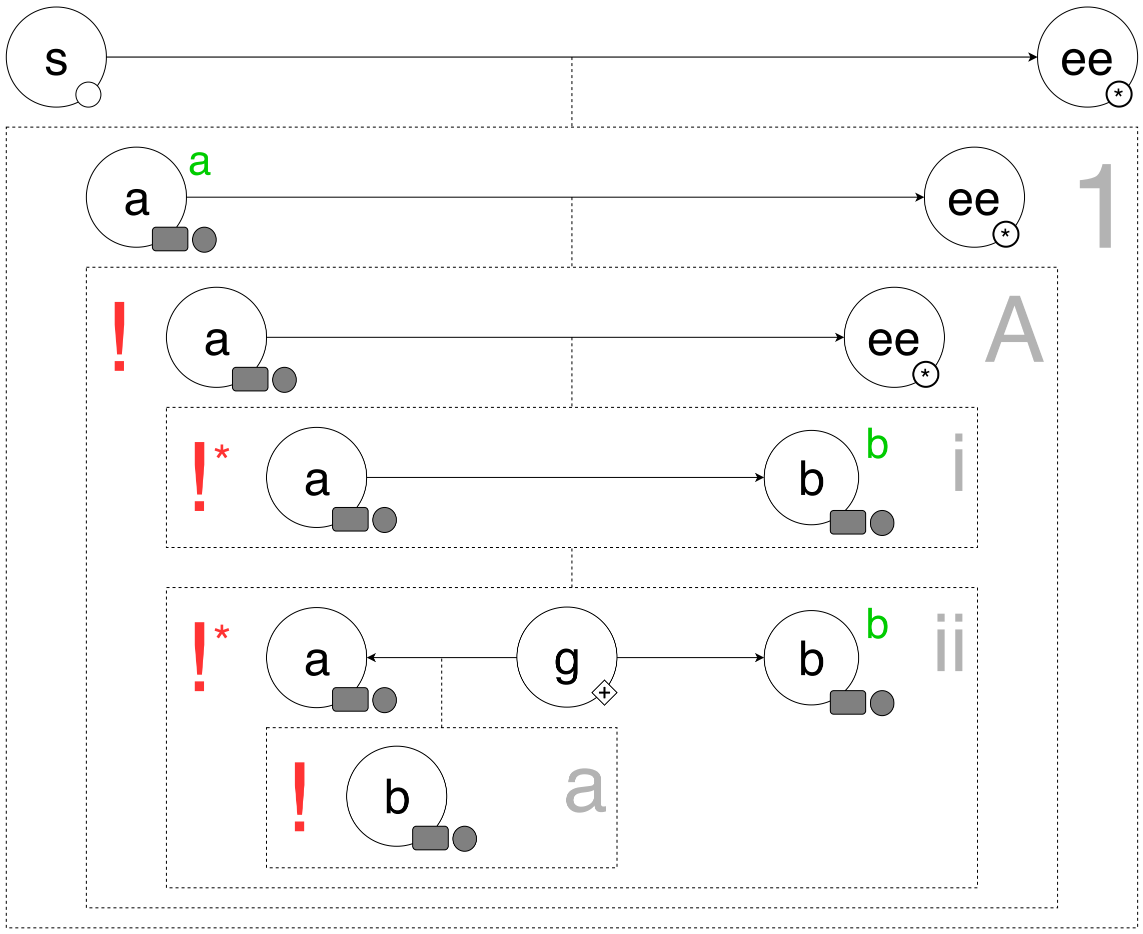


Figure D.2.: DMQL pattern capturing sequential and parallel flow.

D.3. DMQL Pattern Filtering Invalid Conditional Flow

The following DMQL pattern is a modification of the nested *EF* operator pattern where a certain invalid sequence of execution elements is forbidden on fetched paths. Its aim is to show how to produce only valid paths, i.e. paths which map to valid traces.

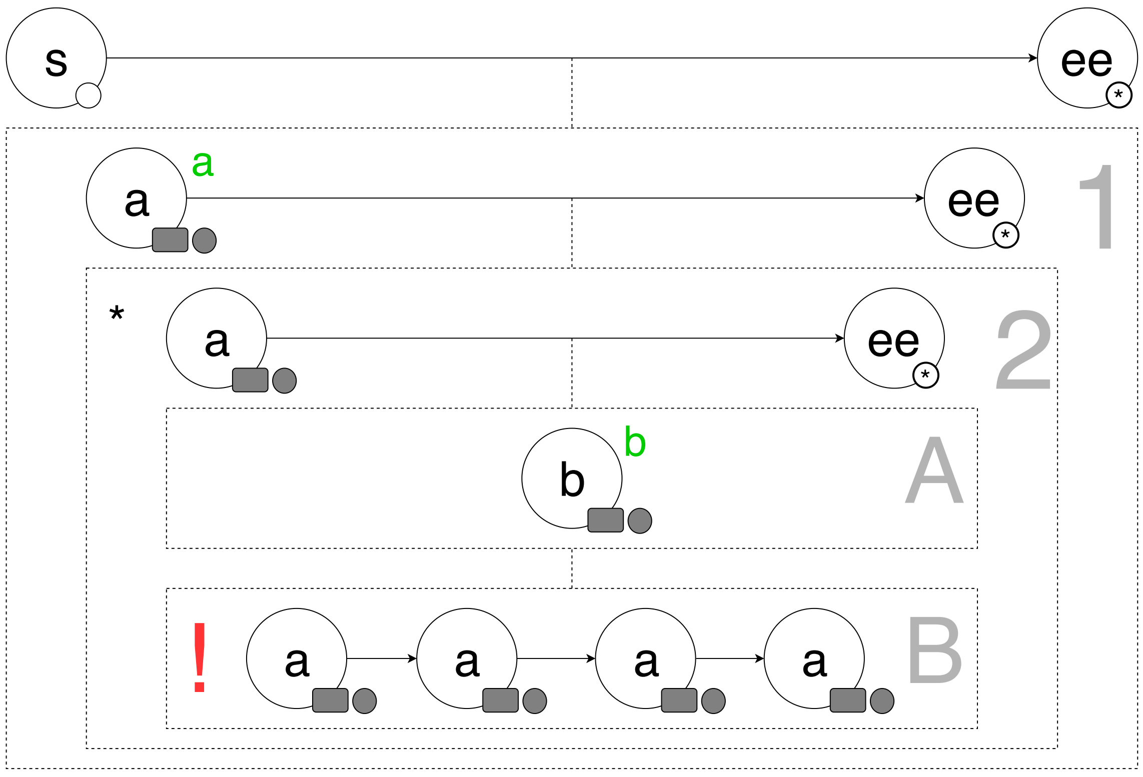


Figure D.3.: DMQL pattern filtering invalid conditional flow.