An Graph-based Approach for Querying Structural Information of Business Processes

Wei Lu

Institut für Architektur von Anwendungssystemen (IAAS)

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studiengang: Informatik

Prüfer: Prof. Dr. Frank Leymann
Betreuer: Dipl.-Inf. Zhilei Ma
begonnen am: 03. März 2008
beendet am: 02. September 2008

CR-Klassifikation: E.1 F2.2 G2.2 H3.3 H4.1
Abstract

Business process specifications can be transformed into attributed directed acyclic graphs. Querying structural information of business process specifications means that find business processes that match the query pattern and its query predicates. In this thesis, we reduce the problem of answering queries against business process specifications to the graph matching problem and present an approach to find exact or approximate answers to such queries. We defined two matching semantics, i.e. exact match and inexact match. We first present a family of stack-based structure join algorithms for an efficient query processing under the exact matching semantic. The worst case time complexity is quadratic in the average size of the query variable bindings. Then we present a new inexact graph matching algorithm to answer approximate queries on process structures by using a new similarity metric called connected component based metric and a new data structure called solution stream. The time complexity of the inexact matching algorithm is polynomial in both the size of the query graph and the size of the process graph in worst case. The algorithms presented in this thesis have been developed for querying business processes in general, as long as the corresponding graph mapping is defined. To evaluate our algorithms we apply them to query BPEL processes by taking into account BPEL-specific characteristics.
## Table of Contents

**Chapter 1: Introduction** ........................................................................................................ 3
  1.1 Structure of the Thesis ........................................................................................................ 4

**Chapter 2: Background And Related Work** ........................................................................ 5
  2.1 Technological Foundation ................................................................................................... 5
  2.2 Requirements on Querying Business Process Specifications ......................................... 10
  2.3 Business Process Queries ................................................................................................. 11
  2.4 Existing Query Languages ............................................................................................... 14
  2.5 Overview of Graph Matching Algorithms ......................................................................... 21

**Chapter 3: Graph Pattern Matching Approach for Query Evaluation** .............................. 33
  3.1 Overview of the Solution ................................................................................................... 33
  3.2 Holistic Twig Joins Approach for Pattern Matching on Trees ......................................... 37
  3.3 Extended Algorithms for Pattern Matching on Directed Acyclic Graphs ....................... 44
  3.4 Approximate Searching Method ....................................................................................... 61

**Chapter 4: Prototype Realization** .................................................................................... 82
  4.1 Applied Technologies ........................................................................................................ 82
  4.2 System Architecture and Implementation ......................................................................... 84

**Chapter 5: Conclusion and future work** .......................................................................... 94

**Bibliography** ....................................................................................................................... 96

**List of Figures** ..................................................................................................................... 101

**Index of Listings** ................................................................................................................ 102

**Index of Abbreviations** ..................................................................................................... 103
Chapter 1: Introduction

A business process (BP) consists of a collection of activities that provide a service or a product that meet the needs of customers. Business process is performed following process model (also called process specification), which is considered more and more as an important enterprise resource. A repository of a company typically stores and maintains a large number of business process specifications. Business analyst or business process designer or other users are oft interested in retrieval structure information of business process specifications based on a given process query. For example, business process designer may enter a portion of a process, and look for business processes, which contain the specified process fragment, to help to design a new process based on existing similar processes. The importance of query languages for business processes has been recognized by Business Process Management Initiative (BPMI) who started a Business Processes Query Language (BPQL) initiative in 2002 [BPQL]. However, no draft standard was published yet.

In this thesis, we propose a new approach for graphically querying business process specification. Given a business process, the most natural representation is to view it as a directed and labelled (attributed) graph whose nodes are activities, whose edges are control links between the activities. We model the specification of a set of business processes as a set of directed attributed graphs (called business process graphs or process graphs).

### Definition 1.1.

The business process searching problem can be characterized as following: Given a business process graph database \( D = \{G_1, G_2, \ldots, G_n\} \) and a query graph graph \( Q \), find all the business process graphs in which \( Q \) appears or approximately appears.

Just as keyword searching matches words against character sequences, process graph searching matches query graph patterns against underlying business process graphs. The challenge is that efficient graph matching algorithms are required in order to answer query in a short time because graph matching algorithms are NP-complete in general [GaJo79]. The main task of this diploma thesis is to develop such efficient algorithms, which are independent on the underlying business process modelling language. Specifically we take a look at Business Process Execution Language for Web Services (BPEL4WS or BPEL), which is the OASIS standard for specifying of business processes by orchestrating Web Services [BPEL]. The generic concept and algorithms for business processes are to be extended to show how it works on BPEL process models by taking into account BPEL-specific characteristics. Based on the concept and the algorithms, a prototype for querying BPEL processes is developed. It accepts a query in form of BPEL-fragment (see definition 2.3), and offers two options to allow user to retrieve either exact matching results or inexact matching results (see definition 2.4, 2.6 and 2.7).
1.1 Structure of the Thesis

The paper is organized as follows:

Chapter 2 covers basic knowledge about Business Process Modelling (BPM), BPEL, business process queries, three existing query languages: SQL [SQL], XQuery [XQuery01], BP-QL [BEK+06] and graph matching algorithms, which are needed to understand the following chapters. Through introduction of BPM and BPEL, we know that business process specification can be represented as a directed acyclic graph (DAG). Therefore, graph matching approach can be used for querying business process specifications. We examine the general requirements on querying business processes. Based on these requirements, we analyze advantages and disadvantages of each of these discussed query languages (SQL, XQuery, BP-QL, our graph based query language) for querying business processes and give the motivation to develop the new graph based query language. The final part of chapter 2 classifiers the graph matching algorithms and gives a brief overview of each category of these algorithms. The selected algorithms are introduced in chapter 3.

Chapter 3 presents both exact and inexact graph matching algorithms for efficient query evaluation. Exact matching algorithms include a holistic twig join algorithm and its extension algorithms for directed acyclic graph (DAG). In order to find approximate answers to a query, we extend the selected exact graph matching algorithms to inexact graph matching algorithms by applying a path based cost model. In order to avoid a big disadvantage of the extended algorithms, an alternative inexact matching method is presented.

Chapter 4 presents implementation of the prototype for querying BPEL-processes. Especially it gives an overview of the system structure, and presents contents and structures of main components.

Finally, Chapter 5 draws a conclusion and provides an outlook proposing suggestions on how to further prosecute the matter.
Chapter 2: Background And Related Work

This chapter introduces the necessary basics and related work that are needed to understand the content of this diploma thesis. First, the basic terms and concepts of BPM and the OASIS standard of Business Process Executable Language (BPEL) are introduced. We show that business process specification can be represented as a directed acyclic graph (DAG), which allows graph matching approach to be used for querying business process specifications. Following that, we analyze the general requirements on querying business process specifications. Section 2.3 presents main characteristics of our graph based query language. We show that the graph matching approach fulfils the requirements of the previous section. After that, section 2.4 analyzes three existing query languages that may be applied for querying business processes, namely SQL, XQuery, BP-QL, and discusses about their application areas, advantages and disadvantages for querying business processes in comparison to the new approach in this thesis. The final part of this chapter gives a brief overview of various graph matching algorithms. We explain why choose or not choose particular algorithms for querying business processes. The selected algorithms are introduced in chapter 3.

2.1 Technological Foundation

2.1.1 Business Process Modelling

A business process consists of a collection of activities and rules prescribing how to navigate activities to process input. We can intuitively understand business process as the step-by-step rules designed to produce a product or service, for example, the loan process in a bank. From technical point of view, business process is also called workflow. Business process modelling (BPM) is about definition and execution of business processes, using standard graphical or XML representations, as a graph structured of activities [Har05].

Business process is performed following the process model, which defines all possible paths through the business process, including the rules that define which paths should be taken and all actions that need to be performed [LeRo00]. It is a template, from which business process is instantiated. When there is no ambiguity, we use these three terms: “business process”, “business process specification” and “business process model” interchangeably in this document.
An important portion of a process model is the definition of the process logic. It involves two major facets: the control flow and the associated data flow [LeRo00]. In this thesis, we focus on the control flow, which defines the execution order of various activities. A business process contains a set of control connectors. A control connector connects two activities and optionally a transition condition can be attached to it. A control connector is only followed if the associated transition condition evaluates to true. There is a start activity in a business process [BEK+06]. Execution of a business process begins with the start activity. There is at least one path between the start activity and each of the rest activities. Typically, an activity of a business process is associated with data. Data flow model the data and the data dependencies between activities. For querying business process specifications data flow is less relevant than control flow because normally user has more interest in structure of business process than the detailed internal data information and process structure can be revealed by control flow. On the other hand, data flow shows data containers of activities and the state of a business process. A transition condition is a predicate in the process data. The data flows within the business process and transition conditions determine which paths should be taken and these are more interesting for monitoring of business process execution.

![Figure 1: A sample business process](image)

DEFINITION 2.1. An activity that is the source of parallel work to be performed is called a **fork activity**. An activity that is the target of parallel work is called a **join activity**. An activity is called join activity if and only if it has more than one incoming control connector. [LeRo00]
Control connectors cannot bind activities in arbitrary manner. There are two restrictions on control connectors of any process model [LeRo00]: First, between two activities there is at most one control connector (called unified property). Second, control connectors must not build loops (called acyclic property), that means graphical representation of a process model must be acyclic. Both restrictions imposed on the metamodel help to avoid ambiguities and improve the comprehension of process models.

**DEFINITION 2.2.** We present the data model of business process for querying business process specifications. We model a business process as a labelled directed acyclic graph with a root node (see definition 2.11 about rooted DAG) called business process graph. If a business process has exactly one start activity, the start activity is mapped to the root node. If a business process has more than one start activity, we create a new root activity and connect the root activity with all start activities. In special case, if there is no join activity in a business process, it can be modelled as a labelled tree (see definition 2.12 about tree and definition 2.13 about join node). Formally, a business process graph is a triple \( g = (G, \lambda, \text{start}) \), where: \( G = (V, E) \) is a directed acyclic graph in which \( V \) is a finite set of activity nodes and \( E \) is a set of edges defined by control connectors; \( \lambda : V \rightarrow L \) (\( L \) is domain of node labels) is a labelling function for the nodes; and start is the start node in \( g \) (also called root).

There are two common ways to represent business processes: XML and notational. In the XML area exist Business Process Execution Language for Web Services (BPEL4WS or BPEL), Business Process Modeling Language (BPML) [BPML], XML Process Definition Language (XPDL) [XPDL], and other approaches. Notational languages include Business Process Modeling Notation (BPMN) [BPMN] and UML activity diagrams [UML]. Among these business process languages, BPEL is the most popular and is a de-facto standard language for modelling and execution of business process.

### 2.1.2 WS-BPEL

Business Process Execution Language for Web Services (BPEL4WS or BPEL) is an OASIS standard to specify and execute business processes based on Web Services [BPEL]. It is a workflow-based language that aggregates services by choreographing service interactions [WCL+05]. The aggregation is recursive, so a BPEL-process is a Web Service itself.

In this thesis, BPEL 2.0 is used as meta-model. In the following, we examine BPEL constructs, which are relevant for this work. A full description of BPEL 2.0 language constructs and semantics can be found in [BPEL].

The major building blocks of BPEL business processes are nested scopes that contain relationships to external partner services (defined by Partner Links), declarations for process data (Variables), various handlers (fault handlers, event handlers, compensation handlers etc.) and most impotently, the
activities to be executed [WCL+05]. In this thesis, we focus on activities, since they define the work to be performed by a BPEL process. Each business process has one main activity. In BPEL, there are two different kinds of activities, basic and structured activities. Basic activities are those which describe elemental steps of the process behaviour, such as the assignment of data to a variable via an assign activity, the invocation of an external partner via an invoke activity or the receipt of requests from an external partner via a receive activity.

Structured activities encode business logic and can contain other basic or structured activities recursively. They prescribe the execution order of the enclosed activities. In BPEL control flow is defined by a combination of structured activities and control links. The following control-flow patterns are defined:

- `<sequence>`, `<if>`, `<while>`, `<repeatUntil>`, and the serial variant of `<forEach>` define sequential control between activities.
- `<flow>` and the parallel variant of `<forEach>` define concurrency and synchronization between activities.
- Deferred choice is provided by `<pick>`.

We introduce following structured activities briefly:

Activities included in a `<sequence>` are performed sequentially in the order in which they appear in the process definition.

The `<pick>` activity is consisted of a set of branches, each containing an event-activity pair. There are two types of event: the `<onMessage>` waits for the receipt of a message and the `<onAlarm>` waits for a timer-based alarm. If an event from the defined events in `<pick>` occurred, the activity associated with that event executes and after that, the other events are no longer accepted by that `<pick>`.

Activities nested in a `<flow>` activity can be executed in parallel. The `<link>` construct is used to define synchronization dependencies between activities enclosed by `<flow>`. Links are declared in a `<flow>` activity. A `<link>` has a unique name among all `<link>` names defined within the same immediately enclosing `<flow>`. Each BPEL activity has the optional containers `<sources>` and `<targets>`, which contain collections of `<source>` and `<target>` elements. Each `<source>` or `<target>` contains a link name that means the enclosing activity is the source or target of the specified link. Links can have a boolean transition condition, which is evaluated after the completion of the link’s source activity and only if the result is true the target will be run. A link must not cross the boundary of a repeatable construct or the `<compensationHandler>` element. A `<link>` declared in a `<flow>` must not create a control cycle, that is, the source activity must not have the target activity as a logically preceding activity [BPEL]. This implies the resulting control graph by flow activity is acyclic.
A `<scope>` provides the context that influences the execution behaviour of its enclosed activities. This behavioural context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler. The process itself is a special case of a scope. Each `<scope>` has a required primary activity that defines its normal behaviour. The primary activity can be a complex structured activity, with many nested activities to arbitrary depth.

Other structured activities just like we known from conventional programming languages are the `<while>` activity which allows the specification of a while loop, or the `<switch>` activity which provides conditional behaviour, or the `<if>` activity provides conditional behaviour.

As described so far, the execution order of activities is defined by structured activities. You can picture the process as a graph with edges between activities that defined possible execution paths. These edges can be (a) the result of structured activities, like the sequence activity, which implies edges between contained activities in the order of their appearance, or (b) defined explicitly by declaring an activity a source or target of a named link. In case (a), sequential control and deferred choice control have quite simple structure, but in case (b) concurrency control defined by `<flow>` can result complex structure, but it must be acyclic (see above the description of flow activity). There may also be forks and joins in the control flow. Forks start multiple paths of execution whereas joins synchronise paths. In general, BPEL business process is modelled as directed acyclic graph (DAG). In special case, if BPEL business process does not contain a join node, which has more than one target element, it can be modelled as a tree.

It has been showed representation of BPEL process as DAG by analyzing BPEL specification. As next, we examine modelling of BPEL process from another perspective. It is well known that BPEL is serialized as an XML file, and an XML document can be modelled as an ordered tree, whose nodes correspond to document elements (or attributes), whose edges represent direct element-subelement relationships. A series of efficient XML pattern matching algorithms were proposed to process XML query in the literature. A question may rise: Why not simply model BPEL file like XML as an ordered tree for the search application. Our goal is to provide a query language, which allows for an intuitive formulation of queries on BPEL specification to search process structure information. If we represent BPEL file the same way as an usual XML file, we lost the business process structure, such representation can not express arrangements and relationships between business activities, which are established by various structured activities. An XML query language based on ordered tree data model such as XQuery is extremely inconvenient for querying BPEL processes. To express even a very simple inquiry about a process execution flow, one needs to write a fairly complex XQuery query that performs an excessive number of joins. Therefore, we adopt a different data model and model ordering between activities according to enclosing structured activity. Another characteristic of BPEL is that the order of the child elements of a BPEL element is not necessarily important. BPEL activities enclosed in a sequence activity are by
definition ordered. The else element must be placed as the last child of an “if” activity. In any other cases, the order of children elements may be presented in any arbitrary order without affecting the behavioural semantic of the BPEL process.

2.2 Requirements on Querying Business Process Specifications

Before we introduce our graph based query language and some existing query languages, it is necessary to present general requirements for querying business process specifications in general and BPEL ones in particular.

**Paths/Graphs extraction:** when querying business processes, users are often interested to retrieve execution paths as answers (as for instance in the query “What should I do to confirm my purchase?”), or qualifying business process graphs as answers (as for instance in the query “Does the travel service provide car renting and flight booking services after my registration? After booking car or flight, can I book some event tickets online?”).

**User friendly:** A business process language such as BPEL usually has a complex syntax. Answering a query like above mentioned is extremely hard when the business process logic is encoded in complex language constructs. In order to ease of querying, it requires a data model that allows to naturally represent the main features and components of a business process, and an equally intuitive query language. As mentioned in section 2.1, a business process can be intuitive represented as a graph of activity nodes, connected by control and data flow edges. It would be much easier if we formulate queries with this intuitive graphical representation.

**Efficient query evaluation:** Business process repositories may store hundred thousands or even more business processes. It is important to answer a query against such large amount business processes in a short time. Therefore, efficient algorithms for query evaluation must be provided.

**Support exact search and inexact search:** A query language for querying business processes should support both exact search and inexact search (see definition 2.7), that allows user to retrieve either exact matching results or inexact matching results of a query according to the desired search type (see definition 2.4 and 2.6).
2.3 Business Process Queries

A business process query includes some user specified selection criteria, which are used to locate particular business processes. Those business processes, which fulfil the selection criteria, are returned as search results in form of process identifier, process location or other representation.

There are two main types of process query languages: structural and behavioural [DeMi07]. Structural query languages allow users to formulate queries about structure of business process specifications while behavioural query languages allow users to ask questions about the possible runs of business processes defined by the specifications. Querying the possible runs of a system is a verification problem [FBS04], and it is NP-hard for simple specifications and undecidable in the general case [NaMc03]. Behavioural query languages are used to identify invariants, execution patterns etc. that is out of the scope of this thesis.

Our graph based query language is a structural query language and is used to formulate queries about business process specifications, for example, does the business process contain these three activities A, B, C and A is executed before B and C.

As mentioned in section 2.1, we model a business process as a labelled directed acyclic graph with a start node. In this thesis, we deal with structural business process query (we call it process query or query in this document for short).

DEFINITION 2.3. A **structural business process query** can be viewed as a business process fragment with activities having value-based predicates. A **business process fragment** is a part of the original business process. It contains a set of activities connected by control connectors. Unlike a business process, it describes an incomplete business process logic. A **structural BPEL process query** is basically a BPEL fragment. A **BPEL fragment** is a XML representation, which contains at least one BPEL activity. The root element of a BPEL fragment must be `<process>` element and the specification of the BPEL activities and the corresponding attributes must be compliant with BPEL standard.

Just like business process, a business process fragment can be represented as a directed graph (called query graph in this context). In order to match with business process graph, a valid query graph must be acyclic, but not necessarily have a root. In general, at each node in the query graph, there is a node predicate on the attributes of the node in question (called query node predicate). Matching between a query node $q$ and a process node $p$ means that predicate of the query node $q$ is satisfied by the process node $p$. For illustration purposes, process node and query node of examples illustrated in this document are labelled with single string attribute and query node predicate is string comparison. Wildcard “*” is allowed to appear in query node label. In the query label string may substitute for zero or more characters in a data string. For example, if a query node labelled with “search*”, then the “*” would substitute for the substring “Train” when matching with a process node labelled with “searchTrain”. For BPEL, different node may have
different attributes according to the node type. Matching of two nodes in BPEL means that both nodes are the same type and all attributes of both nodes satisfy the predicate. If user wants to ignore some attributes (for example portType and inputVariable of an “Invoke” activity) by search, he simply do not specify any value for these attributes, and they will not be compared during the matching between query graph and process graph (semantically equivalent to “*”).

In order to make the concept of our graph based query language understandable, we illustrate an example business process and several queries. Figure 2(a) shows a sample travel booking process and figure 2(b), (c) are examples of twig (see definition 2.12), rooted DAG (see definition 2.11) pattern queries. User may be also interested in finding: Can I begin to search a trip, including searching cars, ships and hotels. This query can be expressed by a twig pattern as shown in figure 2(b). The twig pattern is not contained entirely in the process graph of figure 2(a), because a path “Start Search Trip || Search Ships” is missing. If user wants to retrieve exact matching results (see definition 2.4), the process of figure 2(a) should not be returned as an answer. Otherwise, if an inexact searching (see definition 2.6 and 2.7) is desired and distance from the query to the matching substructure in the business process is less than specified threshold, then the business process is a matching result. Consider the second query in figure 2(c): Can I start to search a trip, then search flights and hotels, after that begin to reserve trip, check possible reservations. This DAG pattern query has three matchings in the process graph of figure 2(a), since the query node “Check Reserve*” matches three process nodes.
Some queries may not have root node. Consider the query: *Is it possible firstly to search cars and hotels, then to check credit card.* This query is depicted in figure 3(a). Such kind of query can be transformed to a rooted DAG pattern by adding a root node and connecting this root node with other nodes without incoming edge, since the selected graph matching algorithms in chapter 3 require a root node in query graph. The newly added query root node is labelled as the root node label of the process graph (see figure 3(b)), if the process root label does not appear in the query graph. Otherwise, we add a new root for the process graph and the new query root is labelled as the label of the new root of the process graph (see figure 3(c)). It can be showed that the modified query graph has the same matching results as the query graph in figure 3(a).
Now let us examine the requirements described in section 2.2 with our graph matching approach. The first requirement about path/graph extraction is obviously fulfilled. It is user friendly, because it is natural and intuitive to represent business process and query as directed graph, and it is quite simple to formulate a query about process structure. What we need are efficient algorithms for query evaluation under exact or inexact matching semantics (see definition 2.4 and 2.6). We present the algorithms in chapter 3.

2.4 Existing Query Languages

Among existing query languages we introduce three representative query languages (SQL, XQuery and BP-QL) which are possible used for querying business processes. In the following, we explain why we choose to introduce these three particular query languages and discuss about their application areas, advantages and disadvantages for querying business processes (cp. with the new approach in this thesis). Through the discussion, we give reasons for developing the new query language presented in this thesis.

2.4.1 SQL

SQL stands for Structured Query Language, and it is an ANSI standard computer language for accessing and manipulating database systems [SQL]. The core of SQL is formed by a command language that allows the retrieval, insertion, updating, and deletion of data stored in a database, and performing management and administrative functions. The most common operation in SQL databases is the query. An SQL query includes a list of columns following the SELECT keyword, the FROM clause, which indicates the source.
table or tables from which the data is to be retrieved, the WHERE clause includes a comparison predicate, and some other statements.

Business process specifications are usually stored in a repository, which is a place (typically a database) where modelling artifacts are stored and maintained. Besides the basic functionality of database (store and query data objects, transaction management etc.), a repository provides some additional features such as checkin/checkout of data object, version control. We then analyse querying business processes with SQL. First, if someone uses SQL directly, he has to know how business processes are stored in database, namely database schema of the business process specification, which is usually not available for a normal user. Second, even user knows database schema, it is still hard and even impossible to formulate SQL queries about business process structure, something like “Can I get a price quote without giving first my credit card details?” The reason is, business logic is encoded in the business process specification, cannot revealed by database schema directly, and SQL is a general purpose query language, not designed for querying complex structured data stored in database directly. Therefore, the first and second requirements of section 2.2 do not meet. SQL is not an appropriate query language for querying business processes.

### 2.4.2 XQuery

XQuery [XQuery01] is a language for querying XML data like SQL for databases. XQuery 1.0 and XPath 2.0 [XPATH] share the same data model, which represents the parsed structure of an XML documents as an ordered, labelled tree in which nodes have identity and may be associated with simple and complex types. XQuery is built on XPath expressions, which are used to locate nodes in XML data. XQuery 1.0’s path expressions are identical to the path expressions of XPath 2.0. A path expression consists of a series of steps, separated by a slash /, or a double slash //. Every step evaluates to a sequence of nodes. Predicate is used to limit the extracted data from XML documents. For instance, given a XML document “books.xml”, the following path expression is used to select all the title elements under the book elements that are under the bookstore element that have a price element with a value that is higher than 30 in the "books.xml" file:

doc("books.xml")/bookstore/book[price>30]/title

XQuery provides a feature called a FLWOR expression that supports iteration and binding of variables to intermediate results. This kind of expression is often useful for computing joins between two or more documents and for restructuring data. The name FLWOR is suggested by the keywords for, let, where, order by, and return. The following FLWOR expression will select exactly the same as the path expression above:

for $x in doc("books.xml")/bookstore/book
where $x\div price>30$
return $x/title$

XQuery is supported by major database engines. BPEL file is essentially an XML document and can be stored in a database. A natural question is why does not query it directly using XQuery? The most important reason has been described in detail in section 2.1.2 that it is very hard to formulate an XQuery query about business process structure. Except that, XQuery only returns document elements, but not paths. XQuery do not meet the first and second requirements of section 2.2 either.

Besides of that, there are some differences between search in XML documents and search in BPEL files, and that must be concerned by applying XML pattern matching algorithms for querying BPEL files:

- Each XML element has single attribute. In BPEL, each activity in general has more than one attribute. Depends on activity type, it may have different attributes. Some of them are mandatory that means the attribute value must not be empty (for example, activity id and type); the others are optional (for example, activity name).
- Some XML pattern matching algorithms require pre-processing. These algorithms are not suitable for querying BPEL files. For XML compare criteria of two nodes are known in pre-processing time, since each node has exactly one label to be compared. In order to be flexible for searching in BPEL files, compare criteria of two activities are unknown before runtime, user can specify single attribute (for example activity type) or combination of several (even all) attributes of a query node as compare criteria of predicate.
- Some XML related algorithms and indexing techniques (such as XR-Tree [JLW+03]) use the fact that many same labelled XML elements may exist. This is not the same for BPEL. The probability that two elements have same label in BPEL is much lower than in XML, if more than one attribute is considered as compare criteria in BPEL.
- Queries in XML query languages are typically path or twig (or small tree) patterns. Queries we presented for BPEL are usually tree or dag patterns.

2.4.3 BP-QL

BP-QL is a new graphical query language for querying business process specifications, with the query being similar to the business process specification, introduced recently in [BEK+05, BEK+06, DM07]. It allows for an intuitive formulation of queries on business process specifications in a distributed cross-organization environment. One may ask, if a query language for querying business processes such as BP-QL already exists, why we need to investigate a new query language. First, BP-QL is involved by BPQ
project of Tel-Aviv University with the intention to contribute to BPQL standard, but it has not been such a standard yet. The characterization of the exact expressive power of BP-QL is still an ongoing research. Second, as we will see BP-QL adopts complete different data model and algorithms. There are some features not covered by our approach (the most important one is in support of fine-grained queries that “zoom-in” on compound nodes), but they need much more computational effort for query evaluation. For many use cases, it is sufficient to use our efficient solution. Third, unlike our new query language, BP-QL is of a totally different representation and data model of the queries in comparison to that of the process specification. This leads to some disadvantages: Users have to learn a new language. Users are not allowed to use existing processes or process fragments as queries. Integration with existing design tools for business process is more difficult than the new query language. Last but not least, in comparison to BP-QL our query language allows more flexible query formulation and provide both exact and inexact search. BP-QL is the most related work on querying business process specifications among these three query languages, so in the following we introduce BP-QL in detail.

**Data Model**

In BP-QL the specification of a BP system is modelled as a set of labelled, directed, possibly recursive nested graphs (in comparison with our data model: flat graphs), including a 'start' node (also called root node) and an 'end' node. The nesting comes from the fact that the operations/services used in a process are not necessarily atomic and may have a complex internal structure. The recursion comes from the fact that a process may call itself indirectly, through calls it makes to other processes.

There are two types of nodes: concrete and compound. Concrete nodes represent process properties, attributes, data elements, and atomic activities. Compound nodes represent compound activities, namely calls of (possibly remote) operations. User may wish to ask coarse-grained queries that consider certain process components as black boxes and allow a high level abstraction, as well as fine-grained queries that “zoom-in” on all (or some of) the process components, possibly recursively. In this thesis, we choose flat graph as data model, because this can reduce time complexity of query evaluation. That means our approach supports only coarse-grained queries but it is sufficient for many use cases.

A formal model based on graph grammars [EEK+99] for systems of processes and BP-QL has been presented in [BEK+06]. This model allows distinguish between query features that can be efficiently supported, and those that incur a prohibitively high cost, or cannot be computed at all. Using this model, the authors explain how to construct a finite and intuitive representation of the (possibly infinite) answers of queries in time polynomial in the size of the specifications.

**BP-QL Query Language**

17
The BP-QL query language presented in [BEK+05, BEK+06, DM07] is based on an intuitive model of business processes, an abstraction of the BPEL specification, along with a graphical user interface that allows for simple formulation of queries over this model. It follows the same design principles that guided commercial vendors in the development of graphical editors for the specification of BPEL processes. It allows user to retrieve paths, and offers facilities for querying at different levels of granularity, and for controlling distributed querying. BP-QL fulfils the first three requirements in section 2.2 and it supports some additional features such as “zoom-in” on compound activity.

At the core of the BP-QL language are business process patterns that allow users to describe the pattern of activities/data flow that are of interest. Business process patterns are similar to the tree- and graph-patterns offered by existing query languages for XML [XPATH], graph-shaped data [COME90, CMW87, PPT95] and the new language proposed in this thesis, but include two novel features. First, BP-QL supports queries with different level of granularity: (1) Fine-grained queries allow to “zoom in” on all the process components (local as well as remote ones); (2) Coarse-grained queries consider process components as black box and allow for high level abstraction. Second, paths are considered first class objects in BP-QL and can be retrieved, and represented compactly, even when involving activities performed on distinct peers.

For querying business processes, BP-QL offers state chart patterns, which play for state charts a role analogous to the one that tree pattern queries play for XML trees. They describe the pattern of execution/data flow that is of interest to the user. In addition to standard state chart notation, the patterns use some special notations as illustrated in the following examples. Similar to the use of ‘/’ and ‘//’ for one and multiple step navigation in XPath and XQuery, they use arrows with single/double heads to denote single or multiple edge paths. For navigation along “zoom in” axis, doubly bounded boxes are used to denote an unbounded “zoom in” into the state’s internal specifications. Selection conditions can be attached to query nodes and edges. Finally, dashed states and edges represent negation.

**BP-QL Query Examples**

Several example queries are illustrated in [BEK+05, BEK+06]. Each query describes a process pattern that a user is looking for.
The query in Figure 4 searches for operations provided by the Alpha-Tours Business Process and (transitively) the compound activities/services that it invokes. The double headed edges inside the behaviour box indicate that activities at any distance from the start/end nodes may qualify; the shape of the node restricts the search to provided operations. The double bounding of the behaviour box denotes unbounded zoom-in. The zoom-in is restricted to activities/services whose specifications reside on the same peer, since the deepSearch attribute is set to local. Setting it to global will extend the search to remote services as well.

Figure 4: A BP-QL query: find provided operations.

Figure 5: (a) Negation                  (b) Path constraints.
The query in Figure 5(a) illustrates the use of negation. It tests whether the users of Alpha Tours are never required to login when searching for flights. Formally, this is expressed by asking: Is there a path of any length that leads to a search, and does not pass through a login request (dashed edges and nodes denote negation). The double boundary of the login box indicates that we refer to requests issued by this process or any of the (possibly remote) services it calls, at any depth of nesting. Query in figure 5(b) retrieves the paths without a login leading to searchFlights that can be expressed by attaching a variable, say x, to the edge, along with the selection condition x ∈ (Σ \ “login”)*.

**Query Evaluation and Compact Representation of Query Results**

For **flat BPs** (BPs with no compound activities), the set of paths between two nodes can be represented by a copy of the sub-graph that connects the nodes. When a BP contains cycles, the number of paths that may match a given (transitive) query edge may be infinite.

For **nested BPs**, a system may contain activity with recursive implementations, hence may have an infinite set of refinements and an infinite number of query results. The solution proposed in [BEK+06] is based on viewing systems and queries as context free graph grammars (abbreviated CFGG) [EEK+99]. A CFGG is a finite set of graphs, where graphs may contain non-terminal symbols, and where grammar rules allow of replacing a non-terminal by a graph from a given finite collection.

The intuition is that, for a system S, the implementation relationships correspond to grammar rules and the system refinements correspond to the graph language defined by the grammar. Similarly, a query q can also be viewed as CFGG whose graph language consists of all the graphs that satisfy the query constraints. Consider the query answer: Instead of constructing explicitly the potentially infinite set of results, one may construct a CFGG that represents it. Specifically, the query answer can be viewed as some kind of “intersection” of the languages defined by the system and query grammars.

In general, the intersection of two CFGG languages may not be a CFGG language [EEK+99]. However, the query specification is sufficiently simple to guarantee the required closure and the intersection belongs to a restricted class of CFGGs called recognizable sets [Cou90]. It is possible use the intersection algorithm presented in [Cou90] to construct a finite representation for the query results. The disadvantage of this solution is that the algorithm of [Cou90] is of high complexity (exponential in the size of the BPs), hence impractical for query evaluation. An important contribution of the work [BEK+06] is that BP-QL queries form a subclass of the recognizable sets for which PTIME solution is possible. A polynomial algorithm (polynomial in the size of the system S with the exponent determined by the size of query) and its correctness proof is available in the full version of the paper [BEK+06].
Implementation of the Query Engine

As mentioned in [BEK+06], the BP-QL prototype system is implemented in Active XML (AXML for short) [AXML]. AXML is a declarative framework that harnesses Web Services for data integration, and works in a peer-to-peer architecture. In AXML documents (extension of XML), some of the XML data is explicit, while other parts are presented intentionally by means of Web service calls. In BP-QL AXML services calls are used to (1) retrieve, when needed, the specifications of remote processes, thus supporting distributed processing, and (2) account for the graph structure of the specification (service calls play here role similar to XML idrefs, with the advantage that they are “traversed” automatically in query evaluation). BPEL documents are wrapped and represented as AXML documents. BP-QL queries are first compiled into XQuery queries for initial processing, followed by post-processing such as validation of zoom-in relationships and extraction of paths for obtaining the desired results. Several problems should be solved during the compilation: First, XQuery does not support path variables. Second, it lacks features that are necessary for expressing (/zoom−in)*, and (/zoom−out)*.

Advantages and Disadvantages by Using BP-QL for Querying Business Processes

The main innovation of BP-QL is in introducing business process patterns that supports navigation along two axis: (1) a novel zoom-in axis, that allows to navigate (transitively) inside process components (local as well as remote ones) and query them at any depth, and (2) the standard path-based axis for retrieval of paths. These features enhance expressive power of BP-QL, but also make the evaluation of queries more intricate than that of flat graphs. Although [BEK+06] propose polynomial time algorithm for query evaluation, but it still has very high complexity. Compare with BP-QL, our approach do not support zoom-in feature, but we achieve better performance for answering query by using efficient graph matching algorithms. Besides of that, where BP-QL focuses primary on path extraction, our approach allows for flexible formulation of query with path, twig or graph pattern. Additional advantages of our approach such as support of exact and inexact search have been mentioned at beginning of this section. Each of both query languages (BP-QL and our graph based query language) has advantages and disadvantages for querying business processes. It depends on concrete situation to choose which of them as business process query language.

2.5 Overview of Graph Matching Algorithms

As motivated in section 2.3 and 2.4, we adopt graph matching approach for querying business processes, and efficient graph matching algorithms are needed for query evaluation. In order to give a clear overview
of various graph matching algorithms, section 2.5.1 first classifies these algorithms. Besides of that, it

gives definitions of some graph types, because graph type has great impact on complexity of algorithms.

From section 2.5.2 to section 2.5.5 different categories of graph pattern matching algorithms are

introduced. For each category of these algorithms, we point out its main character and give the reason, why

we choose or not choose them for querying business processes. For search application, it is important to

evaluate query efficiently, for this reason, time complexity is the main criteria to choose the algorithms.

The selected algorithms for querying business processes are introduced in detail in chapter 3.

2.5.1 Classification of Graph Matching Algorithms and Graphs

In a recent survey [SWG02], methods of pattern matching on graphs are categorized into exact and inexact

matching.

DEFINITION 2.4. The exact matching of query graph and process graph requires a total

mapping from query nodes to process nodes, that means: (1) all query nodes are exactly

matched by their corresponding process nodes (in other words, query node predicates are

satisfied by the corresponding process nodes). (2) The structural (i.e. ancestor-descendant)

relationships between query nodes are satisfied by the corresponding process nodes (each

ancestor-descendant denoted as “//” query edge is mapped to the corresponding path in the

process graph). I.e., we allow the process graph to have nodes in between those found in the

query graph, but their ancestor-descendant relationship still hold in the tree/graph

[CGK05].

EXAMPLE 2.1. An example process graph, a query graph and its exact matching results

are depicted in figure 6. For illustration purpose, both process nodes and query nodes are

labelled with letters, in addition, process nodes with the same label are distinguished by the

affixed numbers.

![Figure 6: Example graph, query, and exact matching results](image.png)
One of the standard concepts in graph matching is graph isomorphism. It is a special case of exact matching, and belongs to exact matching category.

**DEFINITION 2.5.** A **graph isomorphism** from a graph $G$ to a graph $G'$ is a bijective mapping from the nodes of $G$ to the nodes of $G'$ $f: V(G) \rightarrow V(G')$, that preserves all labels and the structure of the edges (two vertices $u$ and $v$ of $G$ are adjacent in $G$ if and only if $f(u)$ and $f(v)$ are adjacent in $G'$). Similarly, a subgraph isomorphism from $G$ to $G'$ is an isomorphism from $G$ to a subgraph of $G'$. We can say graph isomorphism is a special case of subgraph isomorphism.

The definition of graph/subgraph isomorphism seems quite different from the definition of exact matching, but in fact, subgraph isomorphism is such kind of strict exact matching, that each parent-child “/” query edge is mapped to corresponding edge in the process graph.

**EXAMPLE 2.2.** Consider the example process graph and the query graph in figure 6, there is only one subgraph isomorphism solution, namely figure 6(e). The other two solutions illustrated in figure 6(c) and 6(d) are not subgraph isomorphism solutions, because some edges of the query graph cannot be mapped to corresponding edges of the process graph.

In real world applications, we cannot always expect a perfect match between the input and one of the graphs in the database. For example, user does not know the exact process models stored in repository. If user enters a process fragment as a query, those process models similar to the input are expected to be returned. Therefore, inexact graph matching algorithms are needed.

**DEFINITION 2.6.** An **inexact matching** (also called approximate matching or error tolerant matching) allows “approximation” either via a partial mapping from query nodes to process nodes, or by transforming process nodes to establish a total mapping. [CGK05]

**EXAMPLE 2.3.** For query graph in figure 7, there is no exact matching subgraph in figure 6(a), because node $y$ and the corresponding edge have no matching on figure 6(a). Its inexact matching results are in figure 6(c), 6(d), 6(e). It is obviously that exact matching results are included in inexact matching results for given query graph and process graphs.

![Figure 7: Example query](image-url)
DEFINITION 2.7. We define two search types for searching in business processes: (1) exact search returns exact matching process graphs of a given query graph; (1) inexact search returns inexact matching process graphs of a given query graph.

Furthermore, the implementation of matching is classified in [SWG02] as either by traditional (sub)graph-to-graph matching techniques, or by structural join approaches (introduced in chapter 3) which decompose a query into a set of paths and then join the branching nodes of each derived path [AJK+02, VMT04]. The selected exact matching algorithms (see chapter 3) for querying business processes belong to structure join category.

Besides of classification of algorithms, it is also necessary to classify graphs, because for many problems there is no efficient algorithm for general graphs, but it may have efficient algorithms for some special graphs. If a problem is not efficient solvable for a special kind of graphs, then it cannot be efficient solved for more generic graphs. In the following, we present a series of graph definitions that are relevant for this thesis. The order of these definitions is from general graph to more specific graph.

DEFINITION 2.8. A graph $G$ as a data structure is an ordered pair $G := (V, E)$, where $V$ is vertices/nodes set, $E$ is a set of pairs of distinct vertices, called edges.

DEFINITION 2.9. A directed graph $G$ is a ordered pair $G := (V, E)$, where $V$ is vertices/nodes set, $E$ is a set of ordered pairs of vertices, called directed edges. An edge $e = (x, y)$ is considered to be directed from $x$ to $y$; $y$ is called the head and $x$ is called the tail of the edge; $y$ is said to be a direct successor of $x$, and $x$ is said to be a direct predecessor of $y$.

DEFINITION 2.10. A graph is called connected if every pair of distinct vertices in the graph is connected (directly or indirectly). A connected component is a connected subgraph of $G$. Each vertex belongs to exactly one connected component, as does each edge. A directed graph is called weakly connected if replacing all of its directed edges with undirected edges produces a connected (undirected) graph. The maximal weak components are the maximal weakly connected subgraphs. A directed graph is strongly connected if it contains a directed path from $u$ to $v$ and a directed path from $v$ to $u$ for every pair of vertices $u,v$. The maximal strong components are the maximal strongly connected subgraphs.

DEFINITION 2.11. A directed acyclic graph, also called a DAG, is a directed graph with no cycles; that is, for any vertex $v$, there is no nonempty directed path that starts and ends on $v$. A rooted DAG is a DAG satisfying the following properties: (1) there is exactly one vertex (called root), which no edges enter, and (2) each vertex is reachable from the root. A rooted DAG is weakly connected.

DEFINITION 2.12. A (directed) tree [AHU74] is a rooted DAG satisfying the following properties: (1) every vertex except the root has exactly one entering edge, and (2) there is a unique path from the root to each vertex. In this document a twig refers to a small tree.
DEFINITION 2.13. A join node in a directed graph is a node with more than one incoming edges. A join node in business process graph corresponds to a join activity in business process. A rooted DAG which does not contain join node is a tree. Or in other words, if a business process does not contain join activity, it can be modeled as a tree.

DEFINITION 2.14. There are two basic types of trees. An unordered tree is a tree in which the parent-child relationship is significant, but the order among siblings is unimportant. An ordered tree is a tree in which both parent-child relationship and order among siblings are significant.

2.5.2 Subgraph Isomorphism Algorithms

For exact matching we first consider subgraph isomorphism, since it is one of the most commonly used matching conditions. The main problem with subgraph isomorphism detection is the fact that it is an NP-complete problem [GaJo79]. In other words, the time to detect a subgraph isomorphism between two graphs is in the worst case exponential to the number of vertices of these graphs. On the other side, subgraph isomorphism requires exact match of process nodes and edges between query graph and subgraph of process graph that is too strict for querying business processes.

The most common technique to establish a subgraph isomorphism is based on backtracking in a search tree. In order to prevent the search tree from growing unnecessarily large, different refinement procedures such as the one by Ullman [Ull76] (standard algorithm for subgraph isomorphism detection), forward-checking and looking-ahead [HaEl80], or discrete relaxation [KiKa91] have been proposed.

A efficient subgraph isomorphism detection [MeBu00] is described for matching a graph G against a database of model graphs G_1,......,G_N. The basic assumption is that the models in the database are not completely dissimilar. A compact representation of the model graphs is computed offline. Subgraphs that appear multiple times within the same or within different model graphs are represented only once. Such subgraphs will be matched only once with the input. Hence, the computational effort will be reduced. This proposed algorithm has following limitations: It should not be applied to problems dealing with unlabeled, highly connected graphs, to databases of completely disjoint graphs, or to problems where the input graph is considerably larger than the model graphs. The algorithm brings very few benefits for querying business processes. First, business processes have a low degree of similarity in general. Even most business processes are similar the worst case complexity is still exponential with respect to input graph size and model graph size. Second, the algorithm limits flexibility of query formulation due to pre-computation of model graphs. A node in a business process graph may have several attributes and user may want to ignore some of them by search. In preprocessing time, it must determine which attributes are used as compare
criteria, but that is first known in run time. In other words, with this algorithm user is not able to specify which attributes of a query node as search criteria in run time.

2.5.3 Efficient Exact Graph Matching Algorithms for Special Graphs

The exact graph matching problem is NP-complete in general [GaJo79]. However, efficient algorithms exist for some special cases. For example, polynomial-time algorithms are known for planar graph isomorphism and for graphs where the maximum vertex degree is bounded by a constant. A linear-time tree isomorphism algorithm for both labelled and unlabeled trees is presented [AHU74].

We need to know whether we can map a business process onto a special kind of graph, for which efficient matching algorithms exist. As described in section 2.1, a business process is modelled as a directed acyclic graph (DAG). Business process can even be modelled as a tree, if it does not contain join activity. Polynomial algorithms in the size of the graph exist when the graph is acyclic [MEWO89, BFS00]. In a recent work [KNS02], a near-quadratic algorithm (denoted by Nav) in the data graph size is presented for tree and directed acyclic graph (dag) pattern queries on directed acyclic graphs (DAGs) under the exact matching semantics. Besides of that, a family of stack-based algorithms is proposed to handle path, twig, and rooted DAG pattern queries for DAGs, they achieve an optimal runtime complexity quadratic in average size of the query variable bindings. Experimental results show that the stack-based algorithms achieve consistently observed the optimal experimental performance compared to alternative Nav algorithms [CGK05]. The stack-based algorithms are introduced in chapter 3.

2.5.4 Inexact Graph Matching Algorithms

In general, user does not know the exact structure of the data collection to be searched. It is desirable, that the search application is able to retrieve not only results which exact match the query but also results that are similar to the query. If a user specified tolerance is given, only those results whose distance to query under the tolerance are returned. The results are ranked according to their similarity to the query. Exact matches are given first, followed by results with only slight deviations and so on. Inexact (also called error-tolerant or approximate) graph matching methods are needed in this case. An essential problem is how to define similarity or distance of two graphs. In this section, we introduce four similarity metrics and the corresponding inexact matching algorithms, and analyse advantages and disadvantages for each of them.

Graph Edit Distance
Graph edit distance is a common way to measure the similarity between two graphs. A set of edit operations is described by Bunke [Bun97], for example, the deletion, insertion and substitution of nodes and edges, and defines the similarity of two graphs in terms of the shortest (or least cost) sequence of edit operations that transforms one graph into the other. As Bunke et al [MeBu98] mentioned error-tolerant subgraph isomorphism detection is in NP and generally harder than subgraph isomorphism detection. We then consider whether there are efficient inexact matching algorithms for special graphs, e.g. trees and DAGs. As next, we examine inexact matching for trees. If inexact matching problem is not efficient solvable for trees, it cannot be efficient solvable for DAGs either, since DAGs include trees. The exact ordered tree embedding problem has been studied in [RaRa92] and is solvable in polynomial time. However, we are interested in an unordered embedding.

**DEFINITION 2.15.** The problem denoted the **approximate nearest neighbour search (ANN) problem for unordered labelled trees**, is described as the following: Given an integer $\text{DIFF}$, a query tree $Q$ and a database $D$ of trees, the ANN problem is to find all the data trees $T$ in $D$ where $T$ approximately contains $Q$ within distance $\text{DIFF}$. That is, $T$ contains a substructure $T'$ and the distance from $Q$ to $T'$ is at most $\text{DIFF}$.

It has been proved that the ANN problem is NP complete for editing distance [ZSS92]. Therefore, we do not consider edit distance and algorithms based on edit distance as our inexact search solution due to high computational complexity.

**Maximal Common Subgraph Based Similarity Measure**

Another approach to measure similarity of two graphs is based on the maximal common subgraph of two graphs. Relationships between error-tolerant graph matching using graph edit distance and the well-known concept of maximum common subgraph are presented in [Bun97]. The main result of that paper is that, for a particular class of cost functions, maximum common subgraph and graph edit distance computation are equivalent to each other. Hence, any algorithm for maximum common subgraph computation can be used for graph edit distance computation and vice versa, as long as the cost function satisfies the conditions stated in [Bun97]. In close relation with this result, a graph distance measure based on the maximum common subgraph was proposed in [BuSh98]:

$$\quad d(G_1, G_2) = 1 - \frac{|\text{mcs}(G_1, G_2)|}{\max(|G_1|, |G_2|)}.$$ 

In this equation $\text{mcs}(G_1, G_2)$ denotes the maximum common subgraph of $G_1$ and $G_2$ and $|G|$ stands for the number of nodes of $G$. 

27
Maximum common subgraph detection has been addressed in [Mcg82, Lev72, Pel98]. Classical methods for error-tolerant graph matching can be found in [ESFU84, SAFU83, SHHA81, TSFU79]. Most of these algorithms are particular versions of the A* search procedure, i.e., they rely on some kind of tree search incorporating various heuristic look-ahead techniques in order to prune the search space. These methods are guaranteed to find the optimal solution but require exponential time and space due to the NP completeness of the problem. For example: the worst case time complexity of the method described by Levi [Lev72] is $O(nm)^n$, where $n$ and $m$ denote the number of nodes of the two graphs under consideration. Therefore, it is not practical to use these algorithms as our inexact search solution. Besides high complexity, there is another disadvantage for the maximal common subgraph approach. That is it focuses only the maximal matching subgraph, possibly more than one. An example process graph, a query graph and their common subgraphs are depicted in figure 8. If we use maximal common subgraph to calculate the similarity, we get $3/11$ as result. Obviously, this result is not accurate enough to reflect how similar both graphs are, since two other common subgraphs are not considered by the similarity calculation. In general, we should consider which combination of these matching subgraphs provide the maximal partial mapping.

**DEFINITION 2.16.** Assume that $P$ is a process graph, $Q$ is a query graph, $C_1$ and $C_2$ are two common subgraphs of $P$ and $Q$ (or two sub query graphs, which are mapped to $P$). We say that $C_1$ and $C_2$ are **disjoint** or **not overlapping**, if no edge in the query graph is shared by both common subgraphs.

For example, the three common subgraphs between the process graph and the query graph in figure 8 are not overlapping. We present a similarity metric based on these disjoint common subgraphs at the end of this section.

![Figure 8: A sample process graph, a query graph, and their common subgraphs](image)

Note: Only ancestor-descendant relationships between the nodes and subgraphs with at least one edge are considered in this example.
Path Based Similarity Measure

A new approach for approximate search among unordered labelled trees has been introduced in [SWS04]. The distance from a query graph $Q$ to a process Tree $T$ is measured by the total number of root-to-leaf paths in $Q$ that do not appear in $T$. In other words, the similarity of $Q$ and $T$ is measured by the total number of matching paths in $Q$ and $T$. For example, there are two query paths of query graph in figure 7 can find matchings in the process graph in figure 6(a), the matching similarity is $2/3$. The path based similarity measure can be easily extended for rooted DAGs Queries. A possible way is to compute minimal spanning tree for DAG pattern query and count the number of matching paths in it. The reasons for using path based similarity metric are the following:

First, in unordered trees and rooted DAGs, the parent-child relationship or ancestor-descendant relationship is the most significant one to determine structural relationship of nodes that is reflected in edges and paths.

Second, there are efficient decomposition-matching-merging algorithms (also called structural join algorithms), which find all matchings of each query path and merge-join them to produce the result. In section 3.2 and section 3.3, we show that the selected exact matching algorithms, which belong to decomposition-matching-merging category, can be easily extended to perform inexact search by using path-based similarity metric. The complexity is quadratic in the average size of the query variable bindings, same as the selected exact matching algorithms.

The disadvantage of using this method is that the calculated similarity (or distance) of two graphs is not accurate. With the path based similarity metric and the corresponding algorithms, a path in a process graph is returned as a solution path only if it matches an entire path in the query graph, otherwise, it is not returned as a matching path. That means if a query node has no matching process node, then all query paths, which contain the query node, will not find any matchings. The following approximate matching method can be used to address the problem.

Connected Components based Similarity Measure

We present a new graph similarity measure in this section and a corresponding inexact graph matching algorithm to find similar business process graphs to a query graph in section 3.4. As we see in figure 8, there are possible several common subgraphs between a process graph and a query graph. These common subgraphs are weakly connected (see definition 2.10). We call them matching components.

**DEFINITION 2.17.** We define the similarity measure based on a set of disjoint (see definition 2.16) weakly connected matching subgraphs to determine the degree of similarity between two graphs. We call it **connected components based similarity measure**. Assume that $Q$ is a query graph, $P$ is a
process graph, \( \mathcal{C} = \{ \mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n \} \) is a set of disjoint sub query graphs, which are mapped to \( P \). We denote the number of edges of \( \mathcal{C}_i \) by \( | \mathcal{C}_i | \), the reason we use number of edges instead of number of nodes as graph size is that matching edges includes the matching nodes and structural relationships between them. We calculate the matching similarity \( S_{P,Q} = (|C_{\text{max}}| + c \sum (|C_i| | C_i \in \mathcal{C} \setminus C_{\text{max}}))/|Q| \), where \( C_{\text{max}} = \{ C_{\text{max}} \in C \mid |C_{\text{max}}| \) is maximal among all \( |C_i| \)s where \( C_i \in C \}, c \) is a customized connectivity factor. The connectivity factor must be set to equal or smaller than one and greater than zero, the recommended value is from 0.2 to 0.9. We call a matching \( M = \{ \mu: \mathcal{C}_i \rightarrow P_i \mid \mathcal{C}_i \in \mathcal{C} \} \) a maximal matching, if the calculated similarity is maximal among all possible sets of disjoint matching subgraphs.

The reason for multiplication of a connectivity factor for each connected component other than the maximal one is that if matching components are not connected, structural relationships between those components in the process graph are not the same as in the query graph. For example, if we do not multiple the connectivity factor, the similarity for the example in figure 9 is 1, since the three disjoint common subgraphs cover all query edges. But we cannot say that the process graph is a exact matching result for the query graph, since there is no subgraph in the process graph the same as the query graph. If we set the connectivity factor 0.5, then \( S_{P,Q} = (1 + 0.5(1 + 1))/3 = 2/3 \).

![Figure 9: An inexact matching example](image)

We developed a new inexact matching algorithm (see section 3.4) based on connected components based similarity measure for querying business process graphs.

### 2.5.5 Graph Indexing

Given a graph database \( D = \{ G_1, G_2, \ldots, G_n \} \) and a query graph \( Q \), it is inefficient to perform a sequential scan on the graph database and check whether \( Q \) is a subgraph of \( G_i \), since pair-wise comparisons of graphs are usually hard problems. Indexing techniques can be applied reduce the search space and the time complexity especially in large databases. The three basic steps for graph searching in a database are the following:

- Reduce the search space by filtering based on the structure of the labelled graphs (paths, subgraphs).
Several indexing techniques have been developed for graph queries. In the following, we investigate these commonly used graph indexing techniques, and discuss whether each of them brings considerably improvement if we apply it for querying business processes.

In [Mes95, MeBu99] an indexing method for graph and subgraph isomorphism detection is described. It is based on an intensive preprocessing step in which a database of model graphs is converted into a decision tree. At run time, the input graph is classified by the decision tree and all model graphs for which there exists a subgraph isomorphism from the input are detected. If we do not consider the time needed for preprocessing, the computational complexity of the algorithm is quadratic in the number of input graph vertices. In particular, it is independent of the number of model graphs and the number of edges in any of the graphs. However, the decision tree that is constructed in the preprocessing step is of exponential size in terms of the number of vertices of the model graphs. The actual implementation described by the authors is able to cope with a single graph in the database of up to 22 nodes, or up to 30 models in the database consisting of up to 11 nodes each. A business process repository may contain much more business graphs, and a business graph may have much more than 22 nodes. Therefore, this algorithm can not be applied for querying business processes.

GraphGrep, developed by Shasha et al., indexes graphs by enumerating paths up to a threshold length through each graph in a database [SWG02]. An index table is constructed where each row stands for a path and each column stands for a graph. Each entry in the table is the number of occurrences of the path in the graph. Queries are processed in two phases. The filtering phase generates a set of candidate graphs for which the count of each path is at least that of the query. The verification phase verifies each candidate graph by subgraph isomorphism and returns the answer set. GraphGrep has a notable advantage in that its index is not exponential with respect to graph size. However, it is exponential with respect to path length, which is a primary factor in the power of the index.

Yan et al. proposed GlIndex that uses frequent occurring subgraphs instead of paths as index features [YYH04]. Frequent patterns reduce the index space as well as improve the filtering rate. Experimental results show that their technique has 10 times smaller index size than that of GraphGrep, and outperforms GraphGrep by 3-10 times in terms of the candidate answer set size. In a subsequent paper, the authors have extended their idea to process similarity queries [YYH05].

GraphGrep and GlIndex have some common disadvantages, if we apply them to query business processes. First, for these indexing methods pre-processing is required to build the graph index, we get the same problem as mentioned in section 2.5.2 that we know compare criteria of two nodes only after user submit
query, but it should be known in pre-processing time. Second, their index construction requires an exhaustive enumeration of paths or fragments with high space and time overhead, especially when graphs have a low degree of similarity. Consider the index selection criteria in [YYH04]: only frequent fragment is selected to construct high-quality indices. In order to reduce the overall index size, they use size-increasing support constraint as additional selection criterion that means to have low minimum support on small fragments (for effectiveness) and high minimum support on large fragments (for compactness). Let us apply this method to a sample business process graph with 300 nodes and 400 edges. Assume each node of this graph has a distinct label (That is possible if all attributes of activity are used as compare criteria). If we select size 1 to size 4 fragments with minimum support 1, then there are 400 1-edge subgraphs, 79800 2-edge subgraphs, 10586800 3-edge subgraphs. If we set larger fragments with higher support, then we get another problem, that probably no subgraph at all would be retrieved as feature, since the analyzed graphs have very few large common subgraphs. If we apply GraphGrep as indexing method, we get even larger index size than this one. Third, since paths or fragments carry little information about a graph, loss of information at the filtering step appears unavoidable.
Chapter 3: Graph Pattern Matching Approach for Query Evaluation

In this chapter, we present our solution for querying structural information of business processes.

3.1 Overview of the Solution

In this thesis, we provide a generic concept and algorithms for querying business processes. Business process specification can be modelled as an attributed rooted directed acyclic graph (rooted DAG) or as an attributed tree in special case (see section 2.1). A query can be expressed as a tree or a DAG pattern (see section 2.3). Users are allowed to perform exact/inexact search that find matching business processes of a given query under exact/inexact matching semantic (see definition 2.4, 2.6 and 2.7). Efficient exact and inexact graph matching algorithms are needed for query evaluation. The generic concept and algorithms are extended and applied to develop a prototype, which is used for querying BPEL processes stored in a database. In this chapter, we introduce the main components of the prototype and the algorithms.

Before we introduce the components of the prototype, it is necessary to know how does the prototype process a query. Query processing includes the following steps: First, user inputs a query in the form of BPEL fragment (see definition 2.3) and selects the search type. A graphical editor of a BPEL design tool can be used to create such a query. After submission of the query, the query is parsed, then transformed to a query graph. If the query graph is not a DAG, an error message is returned, otherwise, continue to process the query. BPEL Processes stored in database are load in the main memory. After that, each of these processes is transformed to a process graph, and is compared with the query graph by using appropriate algorithms, which are determined by the search type, query graph type and process graph type. If a process graph matches the query graph, process identifier, location of the corresponding BPEL file and the matching part are returned as a matching result. At the end of the query processing, a list of such matching results is returned. The prototype can be extended to show the matching processes in graphical form and highlight the matching nodes, but this does not belong to the work of this thesis.

The prototype is divided into four components: a BPEL database, a transformation component for transformation a BPEL process into a graph, a query evaluation component, and a graphical user interface for input a query and visualization of results. In this thesis, we focus on the query evaluation component. We use BPEL database and main part of the transformation component from another thesis (see [Ger08]...
for description of the both components). We change part of the transformation component in order to adapt the selected algorithms and the needs of the search application. Due to the time limitation, developing a graphical user interface is not in the scope of this thesis. In the following, we give a brief overview of the transformation component and the query evaluation component.

### 3.1.1 Transformation a BPEL Process to a Graph

We have analyzed in section 2.1.2 that BPEL process can be mapped to a DAG. Besides some structured activities, the mapping rules are basically the same as the rules proposed in [Ger08]. We describe only the differences in this section.

For all structured activities except `<sequence>` we place the end node direct after the start node (add an edge from the start node to the end node), all edges between the enclosed nodes of the structured activity and the end node are ignored. The advantage of this change is that every BPEL process without join activity is mapped onto a tree. In figure 10, a sample BPEL process is mapped onto a tree by applying the new mapping rules.

![Mapping a sample BPEL process onto a tree](image)

The rules in [Ger08] consider only activity identifier, name and type as attributes of a basic activity, and an additional attribute for structured activity to determine whether it is a start node. We consider attributes of an activity, which are meaningful for the search, for example, portType and operation for `<invoke>`. If user
wants to search information about a specific attribute of an activity, he can specify a value for this attribute in query. Most attributes are mapped to string type. Attributes of query nodes can be specified with Java regular expression to match the corresponding attributes of process nodes.

We show that the modified mapping rules maps a BPEL business process onto an attributed rooted DAG, whose nodes are activities and whose edges are defined by the mapping rules of the structured activities. As first, we show that each structured activity is mapped onto a rooted DAG, if each of the enclosed activities is mapped to a node (either a basic activity node or a structured activity viewed as a black box). For `<flow>`, edges between the enclosed activities are defined by `<link>`s. `<link>`s declared in a flow activity must not create control cycle [BPEL]. Nodes without incoming links are connected with the start node of the flow activity, which is the root of the resulting flow graph. Other structured activities do not contain `<link>` definition and the corresponding graphs are defined by the rules with regard to the characteristic of the structured activities and possible execution order. Those graphs are per definition rooted and acyclic. As next, we show that the resulting graph is rooted and acyclic, even if structured activities are contained in a structured activity. Each enclosed activity can be viewed as a subgraph; each incoming edge of the enclosed activity is connected with its start activity; each outgoing edge of the enclosed activity is connected with its end activity. There is no such path from the end node to the start node of the enclosed activity, since the enclosing structure activity is mapped to an acyclic graph. Last, we show that there is a root node in each BPEL graph. Each BPEL business process has one main activity. If the main activity is a basic activity, the BPEL process contains only one activity and the corresponding BPEL process graph has exactly one node. If the main activity is a structured activity, the start node of the main activity is the root of the BPEL process graph.

3.1.2 Query Evaluation

Efficient query evaluation is provided by the efficient graph pattern matching algorithms. We use a holistic twig join algorithm and its extended algorithms on DAGs for exact search (see section 3.2 and 3.3). For inexact search, we provide two different methods: (1) one method is based on the modified version of the selected exact matching algorithms and a path based distance metric (see section 2.5.4, 3.2.4); (2) the other one is an algorithms for computing maximal matchings combined with connected components based similarity metric (see section 2.5.4 and 3.4). We choose the second method as default inexact search implementation due to a big disadvantage of method 1 described in section 2.5.4. All these algorithms are independent on the underlying business process modelling language.
The algorithm for query evaluation is presented in Listing 1. During the query processing, we select one of these algorithms according to the search type specified by user, query graph type and process graph type.

For exact search:

- If query graph and process graph are trees, we use TwigStack algorithm [BSK02] (section 3.2.3), which deals with exact matching of a twig query against data collections modelled as trees.
- If query graph is a tree and process graph is a rooted DAG, we use TwigStackD2 algorithm [CGK05] (section 3.3.3), which extend the TwigStack algorithm to handle twig pattern queries on DAGs.
- If query graph and process graph are rooted DAGs, we use DagStackD algorithm [CGK05] (section 3.3.4), which extend the TwigStackD algorithm to handle DAG pattern queries on DAGs.

For inexact search, if we choose the first method (see section 3.4.1), selection of algorithm is similar to above mentioned, but with the modified version. In algorithm ProcessQueryEvaluation, we use the algorithm EvalQueryInexactMatch presented in section 3.4 for inexact search.

Algorithm ProcessQueryEvaluation\((Q, PSet, searchtype)\)

Input:  
\(Q\): a query graph  
\(PSet\): a set of process graphs \(PSet = \{P_1, P_2, \ldots, P_n\}\)  
\(searchtype\): search type

Output: process graphs from \(PSet\), which exact match or inexact match the input query graph \(Q\) depend on the search type

01 if (\(Q\) is not acyclic)  
02 output error message and exit the algorithm  
03 else  
04 if (\(Q\) has no root node)  
05 transform \(Q\) to a rooted DAG through adding a new root node \(r\) to \(Q\), and connecting \(r\) to query nodes without incoming edge  
06 if (\(searchtype == \text{Exact Search}\))  
07 EvalQueryExactMatch\((Q, PSet)\)  
08 else  
09 EvalQueryInexactMatch\((Q, PSet)\)

Listing 1: Algorithm ProcessQueryEvaluation

Algorithm EvalQueryExactMatch\((Q, PSet)\)

Input:  
\(Q\): a query graph  
\(PSet\): a set of process graphs \(PSet = \{P_1, P_2, \ldots, P_n\}\)
Output: process graphs from \( \text{PSet} \), which exact match the input query graph \( Q \)

01 for each \( Pi \) in \( \text{PSet} \)
02 if ((\( Pi \) is a tree) and (\( Q \) is a tree))
03 TwigStack\((Q, Pi)\)
04 else if ((\( Pi \) is a rooted DAG) and (\( Q \) is a tree))
05 TwigStackD2\((Q, Pi)\)
06 else if ((\( Pi \) is a rooted DAG) and (\( Q \) is a rooted DAG))
07 DagStackD\((Q, Pi)\)

Listing 2: Algorithm EvalQueryExactMatch

### 3.2 Holistic Twig Joins Approach for Pattern Matching on Trees

TwigStack algorithm proposed in [BSK02] is a tree pattern matching algorithms, which can be used to find all exact matchings of a query twig pattern (a query tree) in tree structured data. Bruno et al. introduce TwigStack algorithm to computer the exact matching results to a query twig pattern on a XML database, since XML documents are usually modelled as trees and queries are typically twig (or small tree) patterns with some nodes having value-based predicates. The existing methods for twig pattern matching in XML are typically a decomposition-matching-merging process [BSK02, CVZ+02, AJK+02, ZND+01, LIMO01]. The main disadvantage of such a decomposition based approach is that intermediate result sizes can get very large. To address the problem, TwigStack algorithm utilizes a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, and merges the sorted lists of participating element sets altogether, without creating large intermediate results.

#### 3.2.1 Data Model and Numbering Schemes

All exact matching algorithms discussed in chapter 3 use the format (DocId, Start : End, Level) to represent the nodes in the database. This format is also called region encoding. DocId is the identity of the document; Start and End are the start and end positions of the corresponding element in the document (also called preorder rank and postorder rank of the node); Level is the depth of the node in the tree hierarchy (e.g., for the root node, Level = 0).

The advantage of using (DocId, Start : End, Level) to represent nodes is that we can determine the relationships between nodes in a constant time. Suppose that \( x \) and \( y \) are two nodes from an XML tree, we
say that y is an ancestor of x if $y.\text{DocId} = x.\text{DocId}$ and $y.\text{Start} < x.\text{Start} < y.\text{End}$. y is a parent of x if (1) $y.\text{DocId} = x.\text{DocId}$, (2) $y.\text{Start} < x.\text{Start} < y.\text{End}$, and (3) $y.\text{Level} = x.\text{Level} - 1$. For example, in figure 11, the author node $(1, 10 : 12, 4)$ is a descendant of the journal node $(1, 2 : 14, 2)$.

![Figure 11: Tree representation of a sample XML document](image)

**3.2.2 Data Structure and Notation**

In order to ease understanding of TwigStack algorithm, a notation similar to [JWL+03] is used in this thesis. Let $q$ denote a node of twig pattern query, $Q$ denote twig pattern query. Functions $\text{isRoot}(q)$ and $\text{isLeaf}(q)$ examine whether a query node $q$ is a root or a leaf node. Function $\text{isChild}(q_1, q_2)$ examine whether $q_1$ is a child of $q_2$ in the query graph. Function $\text{root}(Q)$ gets root node of $Q$. Function $\text{children}(q)$ gets all child nodes and $\text{parent}(q)$ returns the parent node of $q$. Function $\text{subtreeNodes}(q)$ returns node $q$ and all its descendants.

Two types of data structures (streams and stacks) are used for the twig join algorithm. Given a query tree, associated with each query node $q$ there are: (1) a stream $(T_q)$ with a cursor $(C_q)$ which points to current element of $T_q$ and (2) a stack $(S_q)$, as shown in Figure 12.
The stream $T_q$ contains the positional representations of the process nodes that match the node predicate at the query node $q$ (those match the label), possibly obtained using an efficient access mechanism, such as an index structure. Process nodes in a stream are sorted by their (DocId, Start) values. $T_q[pos]$ is referred to as a process node at position $pos$ of the stream, if any. Each cursor $C_q$ points to certain element in the corresponding stream of query node $q$, hence we also refer to $C_q$ as the process node which $C_q$ points to. Initially, all cursors point to the first element of the corresponding stream. The cursor can move to the next element of the stream $T_q$ (if any) by invoking $C_q\rightarrow advance()$. Attributes of $C_q$ can be accessed by $C_q\rightarrow start$, $C_q\rightarrow end$, $C_q\rightarrow level$.

Each element in the stack $S_q$ consists of a pair: (Positional representation of a matching process node, a pointer to the lowest ancestor in $S_{parent(q)}$). $S_q[pos]$ is referred to as an element at position $pos$ of the stack, if any. Position 0 means the bottom of the stack. The operations over stacks are: empty, pop, push, topStart, topEnd. The last two operations return the start and end coordinates in the positional representation of the top element in the stack. Initially, all stacks are empty. During the computation, each stack $S_q$ may cache some elements before the cursor $C_q$. Each stream node is processed only once, either discarding it or creating a pointer from it to the top node in its parent stack. The cached nodes in stack $S_q$ (from bottom to top) are guaranteed to lie on a root-to-leaf path in the process graph, i.e. each element is a descendant of the element below it. The set of stacks contain a compact encoding of partial and total answers to the query twig pattern. Figure 13 illustrates the stack encoding of answers to a path query for a sample process graph.
3.2.3 TwigStack Algorithm

Algorithm TwigStack [BSK02], which computes all exact matchings of a twig pattern in a process tree, is presented in listing 1. TwigStack is a decomposition-matching-merging algorithm. It finds all matchings of each query root-to-leaf path and these matching paths are merge-joined to produce the answers to the query twig pattern. In listing 1, the main procedure is TwigStack and the others are help methods. In the following, we explain procedure TwigStack and an important help method getNext in detail. The rest help methods in listing 1 are relative small and simple, reader can understand them through the method specification. We assume a process tree and a query tree are given, and streams and stacks are initialized before running the TwigStack algorithm.

Algorithm TwigStack\( (Q, P) \)

Input: \( Q \): a query tree 
\( P \): a process tree 
Output: the input process tree \( P \) and the matching subtrees in \( P \), if \( P \) matches the input query \( Q \) under exact matching semantic

// Phase 1
01 while ¬end(root(Q))
Function getNext(q)
Input:  q: a query node
Output: a query node with minimal descendant extension

01 if (isLeaf(q)) return q
02 for qi in children(q)
03    ni = getNext(qi)
04    if (ni ≠ qi) return ni
05    nmin = minarg Cqi→start
06    nmax = maxarg Cqi→start
07    while (Cq→end < Cnmax→start)
08        Cq→advance
09    if (Cq→start < Cnmin→start) return q
10 else return nmin

Function end(q)
Input:  q: a query node
Output: true or false, that indicates whether all streams associated with query
         leaf nodes are empty

01 Return ∀ qi ∈ subtreeNodes(q): isLeaf(qi) ⇒ end(Cqi)
Procedure cleanStack($S, actStart$)

Input: $S$: stack
   $actStart$: start value of positional representation of a process node

Result: nodes, which will not be participated current path solution, are popped from the stack $S$

01 while ($¬$empty($S$) $∧$ (topEnd($S$) $<$ $actStart$))
02 pop($S$)

Procedure moveStreamToStack($C_q, S_q, p$)

Input: $C_q$: cursor which points to current element of the stream associated with query node $q$
   $S_q$: stack associated with query node $q$
   $p$: a pointer to the lowest ancestor in $S_{\text{parent}(q)}$

Result: node pointed by $C_q$ is moved from $q$’s stream to $S_q$

01 push($S_q, C_q, p$)
02 $C_q$→advance

Procedure showSolutions($q, stackPos, partialPath$)

Input: $q$: a query node
   $stackPos$: stack position
   $partialPath$: partial path from leaf to the current process node in $q$’s stack

Result: Output all path solutions encoded in partial solution stacks in root-to-leaf order

01 $t_q = S_q[stackPos].getProcessNode$  // $t_q$ is the $stackPos$’th process node in $q$’s stack
02 $partialPath$.addFirst($t_q$)  // add the process node at the beginning of the partial path list
03 if (isRoot($q$))
04 output($partialPath$)  // output path solutions from the stacks
05 else  // recursive call
06 for $i = 0$ to $S_q[stackPos].pointer_to_parent$
07 showSolutions(parent($q$), $i$, $partialPath$)

Listing 3: TwigStack Algorithm
Algorithm TwigStack operates in two phases. In the first phase (lines 1-11), some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line 12), these solutions are merge-joined to compute the answers to the query twig pattern. Before we explain TwigStack in detail, it is important to understand how does function getNext work.

Function getNext(q) is called with the root of the query tree and returns a query node q\_x in the subtree q, such that node q\_x has a minimal descendant extension.

Definition 3.1. A query node q has a descendant extension means that there is a solution for the sub query rooted at q composed entirely of the cursor elements of subtreeNodes(q). A query node q\_x has a minimal descendant extension, if the start value of its cursor element C\_q\_x is minimal among all query nodes, which have descendant extensions. In this context, a cursor element is a process node pointed by the cursor of the stream. Note, a cursor always points to the first element of corresponding stream, if there is at least one element in the stream.

Function getNext first traverses down to the left-most leaf node by recursive calls. Starting from the leaf node, it tries to find the highest possible query node with a descendant extension. Given that, all children (all n\_i) have their own descendant extensions after line 4, lines 5-9 determine whether node q has a descendant extension with respect to its children. If common ancestor for all C\_n\_i is found in q, q is returned, otherwise, the child node with the smallest start value n\_min is returned. Note that as long as line 10 is executed and n\_min is returned, the outer recursive calls to getNext will return the same node n\_min all the way up through line 4.

In Algorithm TwigStack, getNext is repeatedly called to find the next processed node q\_act for query root node q (line 2). If q\_act is not query root, in line 4 all elements that are guaranteed not to participate in any new solution are popped from parent(q\_act)’s stack. After that, in line 5 we test whether C\_q\_act participates in a solution. If q\_act is not root and parent(q\_act)’s stack is empty, q\_act doesn’t have an ancestor extension. Therefore, it is guaranteed not to participate in any solution, so q\_act is advanced in line 12 and continue with the next iteration. Otherwise, q\_act has both ancestor and descendant extensions and therefore it participate in at least one solution, q\_act’s stack is cleaned (line 6) and C\_q\_act is pushed in it (line 7). Finally, if q\_act is a leaf node, the stored solutions from the stacks are output with the procedure showSolutionsWithBlocking (lines 8-11), which is a modified version of procedure showSolutions with blocking some path solutions during output.

Now we examine complexity of TwigStack. Consider a query twig pattern q with n nodes, and only ancestor-descendant edges, and a process tree P. Algorithm TwigStack has worst-case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list [BSK02]. Further, the worst-case space complexity of Algorithm TwigStack is the minimum of (i) the sum of sizes of the n input lists, and (ii) n times the maximum length of a root-to-leaf path in P [BSK02]. In the case where the twig pattern
contains a parent-child edge between two elements, algorithm TwigStack is no longer guaranteed to be I/O and CPU optimal [BSK02].

3.2.4 Possible Extensions of TwigStack Algorithm

There are three possible ways to extend TwigStack algorithms.

- TwigStack algorithm presented in Listing 3 returns exact matching (see definition 2.4) results for an input query tree and a process tree that considers only ancestor-descendant query edges. Namely, if two process nodes are mapped to two query nodes connected by a query edge, then these two process nodes are in ancestor-descendant relationship in the process graph. Some applications may also consider parent-child edges in a query tree. For example, if a subtree isomorphism is required, all query edges are considered as parent-child edges. In this case, we need to take levels of query nodes into account. Only procedure showSolutions need to be modified as described in [BSK02].

- TwigStack algorithm can be easily extended to return inexact matchings of a query tree by using path-based metric presented in section 2.5.4. Two modifications are needed: (1) Modify function getNext that allow child stream of returned node to be empty; (2) Count the number of distinct leaf nodes when output path solutions that is the number of matching paths.

- An extended algorithm of TwigStack algorithm for handling rooted DAG query over the rooted DAG pattern process graphs is presented in section 3.3. The extended algorithm can be extended again as above mentioned to handle parent-child edges in a query DAG or inexact matchings.

3.3 Extended Algorithms for Pattern Matching on Directed Acyclic Graphs

Chen et. al extend TwigStack algorithm for processing twig or rooted DAG query over the rooted DAG structured data [CGK05]. They achieve an optimal runtime complexity quadratic in the average size of the query node bindings (e.g., size of streams). Unfortunately, we found through the test that the algorithm sometimes works incorrect. We describe the problem via a concrete example and present our suggestion to fix this problem in section 3.3.4.

3.3.1 Data Structure
**Representation of Rooted DAG**

A rooted DAG G is represented by using a combination of interval encodings on a spanning tree T of G and a predecessor index called SSPI index. In the extended algorithms, both interval encodings and SSPI index are used to determine relationships between two process nodes. Edges, which are in G but not covered by T, are called remaining edges (denoted by \( E_R \)). Namely \( E_R = E - E_T \), assuming E and \( E_T \) denote the edges in G and T respectively. A predecessor index is created for nodes that are reachable from the root via the remaining edges \( E_R \).

In figure 14(a), a tree-cover T of an example rooted DAG G obtained via a depth first traversal of G is illustrated. The tree-cover edges are denoted by solid edges, while the remaining graph edges are depicted by dashed edges. For illustration purpose, a simplified interval encoding format (start : end) is used for this example. A SSPI index for G is showed in Figure 14(b).

![Sample DAG G with node interval encoding on T](image)

![SSPI index](image)

**Figure 14: Illustration of rooted DAG representation**

Formally, a directed graph is represented as \( G = (V, \prec_d) \), where \( V \) denotes all nodes and \( \prec_d \) is a partial order relation between node pairs. Intuitively, each directed edge \( e = \langle a, b \rangle \) in G implies \( b \prec_d a \). The transitive closure of \( \prec_d \) is denoted as \( \prec \). In a rooted DAG G, any path \( p \) can be represented by \( p := (e_i | e_i \in E_T, e_i \in E_R) \), i.e., \( p \) is formed by a mix of tree-cover edges and remaining edges. \( p_{mix} \) denote a path, which
contains at least one remaining edge (i.e., the e_r type). Namely, \( p_{mix} := (e_l e_r^*) e_r^* \), where \( e_r^* \) is the last remaining edge followed by none or more tree-cover edges. The complementary type of such a \( p_{mix} \) path is a path that consists of purely tree-cover edges, denoted as \( p_{pure} := e_t^+ \). If a node \( y \) in \( G \) can be reached from a node \( x \) via a \( p_{pure} \) path, then \( y \prec x \), this reachability information can be revealed from their intervals, namely, \( y: \text{start} > x: \text{start} ^ \wedge y: \text{end} < x: \text{end} \). However, if \( x \) and \( y \) are connected by \( p_{mix} \) paths, then just the node intervals do not suffice to infer the full transitive closure \( \preceq \) of \( G \). The remaining part of \( \preceq \) is denoted as \( \preceq_{\text{rem}} \), which corresponds to all the \( p_{mix} \) paths.

**DEFINITION 3.2.** Suppose \( x \) and \( y \) are the starting and ending nodes of a \( p_{mix} \) path in \( G \). Since \( p_{mix} := (e_l e_r^*) e_r^* \), there must be a node \( w \) along the path such that \( w = \text{child}(e_r) \). \( w \) is a surrogate predecessor of \( y \) if \( w \neq y \). Otherwise, namely if \( e_r \) is incident on \( y \) itself, \( \text{parent}(e_r) \) is an immediate surplus predecessor of \( y \). A **surrogate&surplus predecessor index (SSPI)** is build to hold for each such \( y \) a sorted list of predecessors of both types, denoted \( \text{PL}(y) \), in ascending order by their start interval values. [CGK05]

**EXAMPLE 3.1.** In the SSPI shown in Figure 14(b), \( \text{PL}(s_1) = \{i_1, i_2\} \), \( i_1 \) is an immediate surplus predecessor of \( s_1 \), while \( i_2 \) is a surrogate predecessor of \( s_1 \).

Chen et. al proved that the transitive closure \( \preceq \) can be derived from the interval encodings and the SSPI index. Suppose a process graph \( G \) with rooted DAG pattern has \( n \) nodes and \( m \) edges (\( m \geq n-1 \)). The computation time for building SSPI is \( O(m) \). The total number of predecessors stored in SSPI in the worst case is bounded by \( m \). The computation time for checking ancestor-descendant relationship between two process nodes (via function checkContainment, see below) is \( O(1) \) in the best case and \( O(n) \) in the worst case.

An alternative solution is pre-computation of transitive closure of the process graph. The classic Floyd-Warshall algorithm used for computing the transitive closure is \( O(n^3) \) (\( n \) is the number of nodes in the graph) and the result storage cost is \( O(n^2) \). In addition, a polynomial cost is incurred for maintaining the materialized transitive closure upon data updates. The advantage of this approach is that determine the ancestor-descendant relationship between two process nodes in a constant time. If business processes are updated infrequently or there are many join nodes in business process graphs, pre-computation of transitive closure is a better solution in comparison with the one suggested in [CGK05]. Time and space requirements, update frequency of the stored business processes, characteristics of the business process graphs are factors that should be considered for choosing these two methods.

**Partial Solution Pools**

The extended algorithm utilizes an additional data structure called partial solution pools besides the streams and the stacks. Partial Solution Pools are a data structure, which are utilized to temporarily hold the stack popped nodes to be grown from intermediate partial solutions to full solutions in a bottom-up
fashion. Like stacks and streams, pools are created corresponding to each query node for holding process nodes that match the query node. Two operations are associated with pools:

(1) When pushing a process node \( t_q \) into its stack \( S_q \), the child pool \( \text{Pool}_{ij} \) is swept (i.e., \( q = \text{parent}(q_j) \) in the query graph) to find with the aid of SSPI all such nodes \( t_{qj} \) that \( t_{qj} \) are descendant of \( t_q \).

(2) For each such \( t_{qj} \), a parent pointer from it to \( t_q \) is built to expand the partial solutions headed by \( t_{qj} \) to those headed by \( t_q \).

The sweep operation makes sure that a process node is put into its pool only if it finds descendants in the child pool. This hence guarantees a bottom-up expanding of partial solutions. When a process node is put into the root pool corresponding to the query root node, all the solutions headed by it are output.

### 3.3.2 TwigStackD Algorithm

Algorithm TwigStackD extended from TwigStack algorithm to derive solutions for a twig pattern query for rooted DAGs is presented as follows.

**Algorithm TwigStackD(\( Q, P \))**

Input: \( Q \): a rooted DAG pattern query

\( P \): a rooted DAG pattern process

Output: the input process graph \( P \) and the matching subtrees in \( P \), if \( P \) matches the input query \( Q \) under exact matching semantic

01 while \( \neg \text{end} (\text{root}(Q)) \)
02 \( q_{\text{min}} = \text{getMinSource}(\text{root}(Q)) \)
03 \( \text{missings} = \text{getMissings}(q_{\text{min}}, C_{q_{\text{min}}}) \)
04 if \( \neg \text{isRoot}(q_{\text{min}}) \)
05 \( \text{cleanStack}(\text{parent}(q_{\text{min}}), C_{q_{\text{min}}}) \)
06 if ((complete = \text{sweepPartialSolutionsTSD}(q_{\text{min}}, \text{missings})) \== \text{true})
07 \( \text{cleanStack}(q_{\text{min}}, C_{q_{\text{min}}}) \)
08 if (isRoot(\( q_{\text{min}} \)) \∨ \neg \text{empty}(S_{\text{parent}(q_{\text{min}})})
09 \( \text{moveStreamToStack}(C_{q_{\text{min}}}, S_{\text{parent}(q_{\text{min}}), } \text{pointer to top}(S_{\text{parent}(q_{\text{min}})})) \)
10 if(isLeaf(\( q_{\text{min}} \))
11 \( \text{showSolutionsWithBlocking}(S_{q_{\text{min}}}, 1, \text{null}) \)
12 pop(S_{q_{\text{min}}})
13 else \( \text{advance} \)
14 else \( \text{advance} \)
15 mergeAllPathSolutions()
Function `getMissings(q, tq)`

Input:  
- `q`: a query node
- `tq`: current process node of `q`’s stream

Output:  
- `q`’s children nodes which miss any of the required “inSync” stream nodes or their descendant miss the required “inSync” descendants

01 initialize `missings` as an empty set
02 for `qi` in `children(q)`
03     `pos = 0`
04     if ((`inSync = checkInSync(qi, pos, tq)`) == true)
05         `allInSync = false`
06         while (`pos < size(Tqi) ∧ Tqi[pos].start < tq[end] ∧ ¬ allInSync)`
07             `mi = getMissings(qi, Tqi[pos])`
08             if (¬ empty(mi))
09                 `pos++`
10             else `allInSync = true`
11         if (¬ `allInSync`
12             `missings.add(qi)`
13     else `missings.add(qi)`
14 return `missings`

Function `checkInSync(childq, pos, tparent)`

Input:  
- `childq`: a query node
- `pos`: position of `T_{childq}`
- `tparent`: a process node from the parent stream

Output: true or false, that indicates whether the stream `T_{childq}` has any “inSync” node with respect to `tparent` starting from the position marker `pos`

01 while ((`pos < size(T_{childq})`) ∧ (`T_{childq}[pos].start < tparent[start]`))
02     `pos++`
03     if ((`pos < size(T_{childq})`) ∧ (`T_{childq}[pos].start < tparent[end]`))
04         return true
05 else return false
Function sweepPartialSolutionsTSD(q, missings)

Input:  
q: a query node  
missings: q’s children nodes whose streams miss any of the required “inSync” children/descendant nodes

Output: true or false, that indicates whether every missing child can be complemented by at least one node h in the corresponding partial solution pool

01 for each qi in children(q)
02 for each h in Pool[qi]
03 if (checkContainment(Cq, h) == true)
04     candidateSet[qi].add(h)
05     if (qi ∈ missings)
06         missings.remove(qi) // h < C_q hence qi is no longer missed
07 if (empty(missings)) // if all missings are complemented
08 for each qi in children(q)
09 for each h in candidateSet[qi]
10 expand(q, C_q, h)
11 return true
12 else return false

Function checkContainment(tq, h)

Input:  
tq: a process node from q’s stream  
h: a process node from childq’s stream

Output: true or false, that indicates whether h is descendant of tq in the given process graph

01 found = false;
02 while(¬empty(PL[h]) ∧ ¬found) // PL[h] is a list of predecessors of h in the process graph
03     a = first(PL[h]) // get the first node a in PL[h]
04     if (a.start > tq.start ∧ a.end < tq.end) // interval overlapping
05         return true
06     else if (a.start > tq.end) // a is to the right of tq
07         return false
08     else if (PL[a] == null) // a is to the left of tq and a has no predecessor
09         remove a from PL[h]
10     else if ((found = checkContainment(tq, a)) == false)
11         add the remaining nodes in PL[a] into PL[h]
12         remove a from PL[h]
13 if (empty(PL[h]))
14 remove entry PL[h] from the predecessor lists
15 return found

Procedure expand(q, tq, h)
Input: q: a query node
tq: a process node from q’s stream
h: a process node from childq’s stream
Result: partial solutions headed by h are expanded to those headed by tq, if q is query root, partial solutions headed by tq are output
01 put tq into Pool[q]
02 h.ptr_to_parentPool = tq
03 if (isRoot(q))  // if Pool[q] is the root pool
04 output the solutions headed by tq in the root pool

Listing 4: TwigStackD Algorithm

Like TwigStack, TwigStackD carries out operations in two stages. In the first stage (lines 01-14), solutions to individual root-to-leaf paths are computed, and then they are merge joined in the second phase (line 15) to derive the final query solutions. Similar to TwigStack, in TwigStackD a process node tq (which matches a query node q) is pushed into its stack only if each of q’s descendant qi has a matching process node ti, that ti is “inSync” with tsparent(qi). “inSync” means that a process node tq from a query node q’s stream can find at least one process node ti from stream of q’s child such that ti is descendant of tq. Unlike TwigStack, a descendant of tq of a rooted DAG process graph may exist in two forms: (1) in an input stream and not yet pushed into a stack, or (2) popped from the stack. TwigStack involves only the first case. The reason is that a process node tq (which matches a query node q) in a rooted DAG pattern may have more than one path from the process root node to q, after output the first matching path, tq may have been popped from its stack, but tq may be involved in other matching paths, which are not processed yet. Therefore, tq cannot simply be discarded as in TwigStack, partial solution pools are used to hold those processed nodes. The expanding of partial solutions is carried out only if tq has all of the required descendant types, either located in partial solution pools or still in streams.

Chen et. al. present a series of help functions to resolve difficulties caused by rooted DAG process graph. First, checkInSync is used to check whether the stream Tchildq has any “inSync” node with respect to tsparent starting from the position marker pos (initially set as 0 i.e., the first position, at line 03 in function getMissings). Function getMissings collects all the children nodes, which miss any of the required
“inSync” stream nodes with respect to a specified process node or their descendant streams miss the required “inSync” descendants (lines 04 -12). If $q_i$ is not in the missings returned by getMissings, then the node at the position marker pos in stream $T_{qi}$ is an “inSync” descendant. With the returned missings, TwigStackD calls sweepPartialSolutionsTSD, to check whether every missing child can be complemented by at least one node $h$ in the corresponding child pool (partial solution pool associated with $q$’s child) (lines 01 - 06). Only if this is true (line 07), partial solutions headed by $h$ are expanded to be headed by the new node in Pool[$q$] (lines 08 -10). Now we come back and explain what happens with lines 01-06 in sweepPartialSolutionsTSD: We iterate through each node $h$ in each child pool of the new incoming node $t_q$, and call function checkContainment (line 03) to check whether $h \prec t_q$ by recursively looking up predecessors of $h$ in SSPI. If checkContainment for a node in the partial solution pool of $q_i$ returns true (line 3) and $q_i$ is a missing child, $q_i$ is removed from the missings.

With above mentioned help functions, TwigStackD works as follows: it repeatedly gets a query node $q_{min}$ whose first stream node $C_{qmin}$ has the minimal start value among all stream nodes. Assume that getMissings($q_{min}$, $t_{qmin}$) = $M_{qmin}$. For each query node $q_i \in M_{qmin}$, from lines 11-12 in getMissings we know that the stream $T_{qi}$ either has no “inSync” nodes to $t_{qmin}$ or no such nodes that recursively have “inSync” descendants in the corresponding descendant streams. If Pool[$q_i$] has no such a node connected to $t_{qmin}$ after the sweeping (line 06 in TwigStackD), then $t_{qmin}$ cannot participate in any solution due to the lack of a descendant extension, and it is advanced in line 14. Otherwise, $t_{qmin}$ has a descendant extension composed of the “inSync” nodes in the descendant streams and partial solution pools. Line 8 ensures that $t_{qmin}$ also has an ancestor extension, hence $t_{qmin}$ definitely participates in at least one final solution, and is pushed into its stack (line 9). Finally, a root-to-leaf solution is obtained either when a node is pushed into its leaf stack (lines 10 -11) or put into the root partial solution pool via expand procedure (line 10 in function sweepPartialSolutionsTSD and line 2 in procedure showSolutionsWithBlocking).

Chen et. al. claims that TwigStackD returns all the query answers and guarantees that every solution to every individual root-to-leaf path is merge-joinable with at least one solution to each other root-to-leaf path. Hence, the output individual solutions are no larger than the final solutions to the twig pattern query.

Chen et. al. also presents the complexity of TwigStackD. Given a twig query and a rooted DAG, the worst-case I/O and CPU time complexity of TwigStackD is $O(|q||b_i|^2 + |q||b_i|(d + h_q) + m)$, i.e., max$(m, |q||b_i|((max(|b_i|, d, h_q))))$, where $|q|$ and $h_q$ are the size and height of the twig query, $|b_i|$ is the average stream size of query nodes, $m$ is number of graph edges, $d$ is the number of recursive calls made to lookup predecessors in SSPI (induced from function checkContainment, can be approximated by the diameter of $G$, i.e. the longest length of all shortest paths among data nodes). Note: if we use the pre-computed transitive closure instead of checkContainment with SSPI index, $d$ can be ignored from the time complexity. The worst-case runtime space complexity of TwigStackD is $\min(|b|, p_{max})$, where $|b|$ is total
size of query node streams (|b| can be approximated by |q||b_i|), p_{max} is maximum length of a path in the rooted DAG. This time complexity result is optimal compared to a latest work in [KNS02] whose time complexity is O(n|q||b_i|) for pattern matching on DAGs with a space cost of O(n + m). The dominant factor in the time complexities of algorithms presented here is |b_i|^2 or m (whichever is bigger). Since |b_i| < n and often m < n|q||b_i| for a reasonably large query variable binding size, TwigStackD is more efficient on average compared to the alternative algorithms in [KNS02].

3.3.3 Problems and Trouble Shooting with TwigStackD Algorithm

Through the test, we found that TwigStackD algorithm sometimes does not return all the query answers. In other words, some exact matching results of a query may be “forgotten” by TwigStackD. We demonstrate this issue via a concrete example.

EXAMPLE 3.2. We use the rooted DAG pattern process graph in figure 14(a) (in section 3.3.1) and a twig query in figure 15 as the input of TwigStackD algorithm. Similar to figure 6 in section 2.5.1, both process nodes and query nodes are labelled with letters, in addition, process nodes with the same label are distinguished by the affixed numbers. Streams, which hold the matching process nodes, are also showed in figure 15. Let us start to run TwigStackD. The first five calls of getMinSource(rb) return i, a, r, s, h, the corresponding elements of streams (i_2, a_2, r_2, s_1, h_1) are consumed by the caller TwigStackD (they are neither moved to partial solution pool nor moved to partial solution stack, but simply deleted). That is because they have no missing nodes, thus, SweepPartialSolutionsTSD (line 6) returns true, but they are not root node, and the corresponding parent stacks are empty, so they are advanced from the streams in line 13 one after the other. Partial solution pools and partial solution stacks remain empty. Now it is clear, no exact matching result of the query will be returned, and it is wrong, since there are two exact matchings in the process graph for the given query.
We analyze the problem, and give a suggestion to fix the problem. The main issue is caused by the order of the node to be processed returned by getMinSource. Process nodes in streams are retrieved and processed in their start value order via getMinSource. For a process tree, this means nodes are processed in root to leaf order, but it is not the same for a rooted DAG pattern process graph due to remaining edges, for example, I2 is processed before B1 in example 3.2. If a non-root query node q is the first node retrieved by getMinSource and its current stream element C_q participates the solution, C_q will not be moved from q’s stream into q’s partial solution stack or q’s partial solution pool according to TwigStackD algorithm. Potential solutions, which consist C_q, are lost.

In the following, we give suggestion to modify TwigStackD algorithm and let it work correctly. An important observation is that the concept and principle of TwigStackD algorithm and TwigStack algorithm are very similar, except that, TwigStackD algorithm has to deal with some issues caused by rooted DAG. Instead of getMinSource we use another function getNextExt to get the next highest possible query node with descendant extension. Note that the start value of the cursor element of the query node returned by getNextExt is not necessary minimal among all stream nodes. Function getNextExt(q) (presented in Listing 3) is similar to function getNext(q) (see Listing 1), but with the difference that we apply SSPI index and node interval to determine ancestor-descendant relationship between two process nodes. Newly added codes are lines 12-21. For each query node n_i returned by recursive call with q’s child node we check ancestor-descendant relationship between C_{ni} (the first stream node of n_i) and C_q (the first stream node of q) through function checkContainment2 (see below), if at least one node C_{ni} returned by recursive call is not descendant of C_q, we return n_i, otherwise return q.
We modify function checkContainment, because this time we do not retrieve query node according to the start value of corresponding stream nodes. In order to keep SSPI index unchanged during query processing, we double the predecessor list and associate a boolean variable \textit{isOriginal} for each query node. One predecessor list stores the original predecessors (denoted by \textit{PL}), and they are not changed during the query processing. The other one stores changeable predecessors (denoted by \textit{PL2}). The boolean variable \textit{isOriginal} indicates whether the changeable list differs from the original one. The modified function (called \texttt{checkContainment2}) is presented in Listing 3.

Function \texttt{getMissings} is not used in algorithm TwigStackD2, because a query node returned by function \texttt{getNextExt} guarantees to have a descendant extension.

In order to ease of processing and output final solutions, we develop a new data structure similar to partial solution pools (called solution pools). Before a process node is pushed into its solution pool, we create a new object (we call it pool element) with the type \texttt{PoolItem}, which is a data structure consisting of a process node, a query node and a map which matches the query child node with references to the corresponding child pool elements. During the processing, pool items, which are possible part of final solutions, are created and pushed into the solution pools. If a process node of a pool element in parent pool is a process node of a pool element in child pool, a reference from the parent pool element to the child pool element is added. In the final stage of TwigStackD2, final solutions are output directly from the solution pools. There is no need to merge join of paths. Figure 16 shows solution pools for the query graph in figure 15, after running TwigStackD2 with figure 14(a) as the input process graph, exact matching results are encoded in the solution pools.
We present the TwigStackD2 algorithm and its sub procedures and functions as below:

Algorithm TwigStackD2(q)

Input:  
Q: a rooted DAG pattern query  
P: a rooted DAG pattern process

Output:  
the input process graph P and the matching subtrees in P, if P matches the input query Q under exact matching semantic

// Phase 1
01 while ¬end(root(Q))
02 qact = getNextExt(root(Q))
03 if (¬isRoot(qact))
04 cleanStack(parent(qact), C_{qact}→start)
05 if (isRoot(qact) ∧ ¬empty(S_{parent(qact)}))
06 cleanStack(qact, C_{qact}→start)
07 sweepPartialSolutionsTSD2(qact, C_{qact})
08 moveStreamToStack(C_{qact}, S_{qact}, pointer to top(S_{parent(qact)}))
09 if(isLeaf(qact))
10 \( \text{partialPath} = \text{createEmptyNodesList()} \)
11 \( \text{processPathSolutions}(q_{act}, 0, \text{null}, \text{null}, \text{partialPath}) \)
12 \( \text{pop}(S_{q_{act}}) \)
13 else \( C_{q_{act}} \rightarrow \text{advance} \)
   // Phase 2
14 \( \text{outputAllExactMatchingSolutions()} \)

**Function getNextExt(q)**

Input: \( q \): query node
Output: highest possible query node in the query graph with a descendant extension

01 if (isLeaf(\( q \))) return \( q \)
02 for \( q_i \) in \( \text{children}(q) \)
03 \( n_i = \text{getNextExt}(q_i) \)
04 if \( (n_i \neq q_i) \) return \( n_i \)
05 \( n_{\text{min}} = \text{minarg}_{n_i} \ C_{n_i} \rightarrow \text{start} \)
06 \( n_{\text{max}} = \text{maxarg}_{n_i} \ C_{n_i} \rightarrow \text{start} \)
07 while \( (C_q \rightarrow \text{end} < C_{n_{\text{max}}} \rightarrow \text{start}) \)
08 \( C_q \rightarrow \text{advance} \)
09 \( \text{returnChildNode} = n_{\text{min}} \)
10 if \( (C_q \rightarrow \text{start} < C_{n_{\text{min}}} \rightarrow \text{start}) \)
11 return \( q \)
12 else
13 \( \text{containsAllChildren} = \text{true} \)
14 for each \( n_i \)
15 if (!\( \text{checkContainment2}(C_q, C_{n_i}) \))
16 \( \text{containsAllChildren} = \text{false} \)
17 \( \text{returnChildNode} = n_i \)
18 if \( (\text{containsAllChildren}) \)
19 return \( q \)
20 else
21 return \( \text{returnChildNode} \)

**Function checkContainment2(t_q, h)**

Input: \( t_q, h \): two process nodes with positional representation to be compared
Output: a boolean value indicates whether $t_q$ is ancestor of $h$ in the process graph

01 if ((h.start > $t_q$.start) ∧ (h.end < $t_q$.end)) // interval overlapping
02 return true
03 else
04 found = false;
   // check whether the second predecessor list of $h$ has been modified by the program
05 if (!h.isOriginal)
   // copy predecessors from the first predecessor list of $h$ to the second predecessor list of $h$
06 $h$.PL2[] = $h$.PL[]
07 $h$.isOriginal = true
08 while(!empty($h$.PL2[])) ∧ ¬found)
09 $a$ = first($h$.PL2[]) // get the first predecessor in $h$.PL2[]
10 if (((a.start > $t_q$.start) ∧ (a.end < $t_q$.end)) // interval overlapping
11 return true
12 else if (a.start > $t_q$.end) // a is to the right of $t_q$
13 return false
14 else if (a.PL[] == null) // a is to the left of $t_q$ & has no predecessor
15 remove a from $h$.PL2[]
16 if ($h$.isOriginal)
17 $h$.isOriginal = false
18 else if ((found = checkContainment($t_q$, a)) == false)
19 add the remaining nodes in a.PL2[] into $h$.PL2[]
20 remove a from $h$.PL2[]
21 if ($h$.isOriginal)
22 $h$.isOriginal = false
23 return found

Procedure sweepPartialSolutionsTSD2($q$, $t_q$)

Input: $q$: query node
   $t_q$: current process node in q’s stream
Result: create a new pool element $t_qPI$ with $q$ and $t_q$, and put it into q’s solution pool. Then we
sweep q’s parent pool, if a process node of a parent pool element parentPI is ancestor of $t_q$, we
add a reference from parentPI to $t_qPI$. After that, we sweep q’s children pools. For each child
pool Pool[$q_i$] (i.e., $q=parent(q_i)$ in the query tree) find all such nodes $t_{qi}$ (consisted in the child
pool item $t_{qi}PI$) that $t_{qi}$ is descendant of $t_q$. For each such $t_{qi}PI$, we add a link from $t_qPI$ to $t_{qi}PI$
to expand the partial solutions headed by $t_{qi}PI$ to those headed by $t_qPI$. 
01 create a new pool element \( tqPI \) with \( q \) and \( t_q \), then put it into \( \text{Pool}[q] \)
02 for each \( \text{parentPI} \) in \( \text{Pool[\text{parent}(q)]} \)
03 \hspace{1em} if (checkContainment2(\( \text{parentPI}.\text{getProcessNode}, t_q \)) = = \text{true})
04 \hspace{1em} \text{parentPI}.\text{add_ptr_to_childPool}(\text{tqPI})
05 for each \( q_i \) in \( \text{children}(q) \)
06 \hspace{1em} for each \( tqiPI \) in \( \text{Pool}[q_i] \)
07 \hspace{2em} if (checkContainment2(\( t_q, tqiPI.\text{getProcessNode} \)) = = \text{true})
08 \hspace{2em} \text{tqPI}.\text{add_ptr_to_childPool}(\text{tqiPI})

Procedure processPathSolutions(\( q, stack\text{Pos}, t\text{childq}, childPI, \text{partialPath} \))

Input:  
\( q \): query node
\( stack\text{Pos} \): stack position
\( t\text{childq} \): positional representation of \( q \)’s child
\( childPI \): a pool element in \( q \)’s child pool
\( \text{partialPath} \): partial path from leaf to the current process node in \( q \)’s stack

Result: Output path solutions encoded in partial solution stacks in root-to-leaf order and store these path solutions in partial solution pools, expand solutions for every processed node \( t_q \) by sweeping its parent pool.

01 \( t_q = S_q[\text{stackPos}].\text{getProcessNode} \)  // \( t_q \) is the \( \text{stackPos} \)th process node in \( q \)’s stack
02 \( childPI = \text{expandStackPathSolution}(q, t_q, t\text{childq}, childPI, \text{partialPath}) \)
03 \( \text{partialPath}.\text{addFirst}(t_q) \)  // add the process node at beginning of the partial path list
04 if (isRoot(\( q \)))
05 \hspace{1em} output(\( \text{partialPath} \))  // output path solutions from the stacks
06 else  // recursive call
07 \hspace{2em} for i = 0 to \( S\text{qact}[\text{stackPos}].\text{pointer_to_parent} \)
08 \hspace{3em} processPathSolutions(\( \text{parent}(q), i, t_q, \text{childPI}, \text{partialPath} \))

Function expandStackPathSolution(\( q, t_q, t\text{childq}, childPI, \text{partialPath} \))

Input:  
\( q \): a query node
\( t_q \): a process node in \( q \)’s stack
\( t\text{childq} \): a process node from \( q \)’s child stack
\( childPI \): a pool element from \( q \)’s child pool
\( \text{partialPath} \): partial path from leaf node to \( t_q \)

Return: a previous created pool element associated with \( q \) and \( t_q \)
01 retrieve a pool element tqPI with (tqPI.getProcessNode == tq) from the Pool[q]
    // tq and tchildq are part of the path solution, therefore, we add reference for the corresponding
    // pool elements.
02 tqPI.add_ptr_to_childPool(childPI)
03 return tqPI

Procedure outputAllExactMatchingSolutions()
Return: Output all exact matching results stored in solution pools

01 for each trPI in Pool[root]  // root is the root node of the given query graph
02 output solutions rooted at pool element rPI

Listing 5: TwigStackD2 Algorithm

The key ideas of TwigStackD2 algorithm are the same as the TwigStack algorithm. In line 2, getNextExt retrieves the highest possible query node \(q_{act}\) with a descendant extension. If \(q_{act}\) is not query root node and the stack associated with parent(\(q_{act}\)) is empty (line 5), it guarantees not to participate in any solution and it is advanced in line 13. Otherwise, \(q_{act}\) has both ancestor and descendant extensions and therefore it participates in at least one solution, so it is pushed into the stack. Compare TwigStackD2 with TwigStack, the main part of both algorithms are same besides the following differences.

- Before a process node \(C_{q_{act}}\) is pushed into the stack, we call procedure sweepPartialSolutionsTSD2 to expand solution pools. We create a new pool element tqPI with \(q_{act}\) and \(C_{q_{act}}\), then push it into Pool[\(q_{act}\)] (line 1). Pool element associated with \(C_{q_{act}}\) is unique in Pool[\(q_{act}\)], since each stream node is processed at most once in TwigStackD2. Then we sweep the parent pool Pool[parent(\(q_{act}\))] (lines 2-4). We iterate through each pool element parentPI in Pool[parent(\(q_{act}\))] (line 2), and call function checkContainment2 to check whether the process node consisted in parentPI is an ancestor of the process node consisted in tqPI (line 3). If this is true, we add a link from parentPI to tqPI to expand the partial solutions headed by tqPI to those headed by tQPI (line 4). After that, we sweep \(q_{act}\)'s children pools (lines 5-8). We check ancestor-descendant relationship between tqPI and each child pool element by using function checkContainment2, and possibly add a reference between them depending on the boolean value returned by checkContainment2. The reason to sweep the parent pool and children pools are that a new incoming process node of a rooted DAG may be part of some matching paths, which have been popped from the stacks and stored in solution pools.
• When a process node $C_{q_{act}}$ is pushed on the stack $S_{q_{act}}$, where $q_{act}$ is a leaf node of the query tree, the query path solutions stored in the stacks are processed by procedure \texttt{processPathSolutions}, which is similar to the procedure \texttt{showSolutions} in Listing 1. In procedure \texttt{processPathSolutions}, process nodes are processed in leaf-to-root order. For each such a process node $t_q$ and its child node $t_{child_q}$ (which was processed by a previous call of \texttt{processPathSolutions}), procedure \texttt{expandStackPathSolution} is called in line 2 to add a reference from $t_qPI$ (pool element associated with $t_q$) to $child_qPI$ (pool element associated with $t_{child_q}$), since $t_q$ and $t_{child_q}$ is part of the path solution. Note that those pool elements have been created and stored in the solution pools in procedure \texttt{sweepPartialSolutionsTSD2}. The rest part of procedure \texttt{processPathSolutions} is the same as procedure \texttt{showSolutions}, namely, process the encoded path solutions through recursive call of self. The path solutions encoded in the stacks are output one after the other (line 5) and they are stored in the solution pools (line 2).

• After processing all path solutions, all exact matching solutions are encoded in the solution pools with a tree like data structure. Therefore, there is no need to merge-join the path solutions. In the second phase of TwigStackD, we output all exact matching results with Procedure \texttt{outputAllExactMatchingSolutions}, which outputs solutions rooted at each root pool element. Note that, there may be more than one exact matching solution for a root pool element.

3.3.4 DagStackD

DagStackD algorithm is extended from TwigStackD algorithm for handling rooted DAG pattern queries [CGK05]. The basic idea is that for a join node $v$ in a rooted DAG query, e.g. node with more than one incoming edge, the AND-matching semantic (see definition 3.4) must be enforced, that means, it satisfies all incoming constraints of $v$. DagStackD works as follows. First, we build a minimal spanning tree for the rooted DAG query, and conduct TwigStackD2 algorithm to find the solutions for the minimal spanning tree. Then, for each join node $q_{m}$ in the rooted DAG query, enforce the AND-matching semantic by checking whether each of its bindings $t_{qm}$ has also participated in at least one solution to each other query path in the rooted DAG pattern that also includes $q_{m}$. If not, filter it out similar to what is described in [KNS02]. The time complexity of DagStackD is that of TwigStackD2 with the addition of the complexity for joining the solution bindings at each query merge node among intersecting query paths. Its complexity is still in quadratic in the average size of the query variable bindings. In addition, Chen et al. introduce a novel pre-filtering technique that can prune the nodes to be put in the streams based on the structural pattern constraints of a query. However, the experiments show that the pre-filtering technique only takes effect for large graph, i.e. graph with more than 25000 nodes [CGK05]. Business process graphs are not so large in general, so we do not consider this pre-filtering technique in this thesis.
3.4 Approximate Searching Method

In this section, we present a new inexact matching algorithm for querying business process graphs by using connected components based similarity metric (see definition 2.17).

3.4.1 Matching Semantics

We introduce matching semantics proposed in [KNS02] and some related definition, which are needed to understand our inexact matching algorithm presented in section 3.4.3.

Definition 3.3. Let $V_P$ ($E_P$) be a set of nodes (edges) occurring in a process graph and $V_Q$ ($E_Q$) be a set of nodes (edges) occurring in a query graph. A mapping $\mu: V_Q \rightarrow V_P \cup \{\perp\}$ assigns a query node to a process node or a new symbol $\perp$ (called null). If $\mu(v) \neq \perp$, then $\mu(v)$ must satisfy the predicate of the query node $v$, we say that the assignment $\mu$ is defined for the query node $v$, or that $v$ is bound. Otherwise, $\mu$ is undefined for the query node $v$. An assignment is total if it is defined for all nodes in $V_Q$ and partial otherwise. Each edge corresponds to a binary relation between two nodes, that is $l^Q = E_Q \subseteq V_Q \times V_Q$ or $l^P = E_P \subseteq V_P \times V_P$. An assignment satisfies an edge constraint $ul^Qv$ if $\mu(u) \in l^P(\mu(v))$, i.e., the relation in the process graph contains the pair $(\mu(u), \mu(v))$. An assignment $\mu$ is a strong matching if all query nodes are bound (total assignment) and $\mu$ satisfies every edge constraint in the query graph $Q$, while a weak matching allows some query nodes are unbound (partial assignment) and whenever a weak matching is defined for query nodes of a edge constraint, it has to satisfy the edge constraint. [KNS02]

In fact, a strong matching is an exact matching (definition 2.4) and a weak matching is an inexact matching (definition 2.6). Besides weak matching, two other matching semantics are defined, namely, AND-matching and OR-matching are defined [KNS02].

Definition 3.4. In a query graph $Q$, the edge constraints of the form $ul^Qv$ are also called the incoming constraints of the query node $v$. A matching $\mu$ is an AND-matching if it satisfies all incoming constraints of $v$ whenever $\mu(v) \neq \perp$. A matching $\mu$ is an OR-matching if it satisfies some incoming constraints of $v$ whenever $\mu(v) \neq \perp$. [KNS02]

The sets of strong, weak, AND, OR-matchings of $Q$ over $P$ are denoted as $Mat^s_P(Q)$, $Mat^w_P(Q)$, $Mat^\wedge_P(Q)$, $Mat^\vee_P(Q)$. It is obviously that $Mat^\wedge_P(Q) \subseteq Mat^\vee_P(Q) \subseteq Mat^\wedge_P(Q) \subseteq Mat^\wedge_P(Q)$. For $\sigma \in \{s, \wedge, w, \vee\}$, $MMat^\sigma_P(Q)$ is defined as the set of maximal elements of $Mat^\sigma_P(Q)$.

Note that, a query node is allowed to assign a process node under OR-Matching semantic, if at least one incoming constrain is satisfied. Intuitively, when computing OR-matchings, we view the query graph as a description of a set of possible paths which to explore the process graphs and to collect as much
information as possible. For querying business processes, we use AND-matching semantic for exact search (mentioned in section 3.3.4) and OR-matching semantic for inexact search.

There is no difference between AND- and OR-Matchings for a given query tree, since every query node (except root) has exactly one incoming edge. If a query node \( v \) in the query tree has no matching process node \( (\mu(v) = \bot) \), all descendants of \( v \) are undefined under both matching semantics. For a rooted DAG pattern query, if all parents of a query node \( v \) are undefined, then \( v \) and all descendants of \( v \) are undefined either.

Example 3.3. An example for matchings between a process graph in figure 17(a) and a query DAG in figure 17(b) under AND- and OR-Matching semantics is illustrated. The matching results are showed in table 1. Matching IDs showed in table 1 have no special meaning. It is simply an unique identifier for each assignment.

![Process graph and Query graph](image)

Figure 17: An example for subgraph matching under AND- and Or-Matching Semantics

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Matching ID</th>
<th>o</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Level-1-1</td>
<td>o₁</td>
<td>x₁</td>
<td>y₁</td>
<td>z₁</td>
<td>a₁</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>Level-1-2</td>
<td>o₁</td>
<td>x₁</td>
<td>⊥</td>
<td>⊥</td>
<td>a₂</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>Level-1-3</td>
<td>o₁</td>
<td>x₂</td>
<td>⊥</td>
<td>⊥</td>
<td>a₁</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>Level-1-4</td>
<td>o₁</td>
<td>x₂</td>
<td>⊥</td>
<td>⊥</td>
<td>a₂</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>Level-1-5</td>
<td>o₂</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>a₂</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td>OR</td>
<td>Level-1-1</td>
<td>o₁</td>
<td>x₁</td>
<td>y₁</td>
<td>z₁</td>
<td>a₁</td>
<td>⊥</td>
<td>⊥</td>
<td>c₁</td>
<td>⊥</td>
<td>f₁</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>Level-1-2</td>
<td>o₁</td>
<td>x₁</td>
<td>y₁</td>
<td>z₁</td>
<td>a₂</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>Level-1-3</td>
<td>o₁</td>
<td>x₁</td>
<td>y₃</td>
<td>z₂</td>
<td>a₂</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
</tr>
</tbody>
</table>
Each of these matchings presented in Table 1 starts to assign a process node to the query root node (called level-1 matching). If root is undefined, then all query nodes are undefined under each of these matching semantics defined in [KNS02]. These matchings are overlapping (see definition 2.16), since at least query root node is assigned for each of them. Therefore, they cannot simply combined to compute the similarity between the query graph and the process graph.

**Definition 3.5.** For a node q in a given query DAG, there may be more than one root-to-q path. We call the root node level-1 node, and the query node q whose longest path from the root node to q consists of i-1 edges a level-i node. A collection of query nodes is sorted by level order that is a node at level i is before level j if i < j. We call an OR-matching a level-i OR-matching (or level-i OR-assignment), if it starts to assign a matching process node to query root node. An OR-matching is called a level-i OR-matching (or level-i OR-assignment), if it starts to assign a matching process node p to a query node q, at level i in the given query graph. A level-i OR-assignment maps a query subgraph rooted at a query node qi at level i to a substructure of the process graph under OR-matching semantic. In the following, we call it level-i matching (or level-i assignment) for qi for short, since only OR-matching semantic is used for our inexact matching algorithm. The first assign [qi/p] of a level-i OR-matching is called start assign, where qi is the root node of query subgraph to be assigned. If qc (at level c) is a child node of qi in the query graph, a level-c assignment for qc is called child assignment of qi. A level-i OR-assignment for a query node qi is maximal, if it contains maximal number of matching edges among all level-i OR-assignments for qi. Similarly, we define Level-i matching for other matching semantics proposed in [KNS02].

Only maximal assignments are interested, since they contain maximal information. Note that there may be more than one maximal assignment with the same number of matching edges or nodes for two graphs. In example 3.3, two level-1 OR-assignments with ID “Level 1-1” and “Level 1-4” are the maximal level-1 OR-assignments. A maximal matching (see definition 2.17) between two graphs may be a combination of several such level-i OR-assignments.

**Example 3.4.** A maximal matching for graphs in figure 17 (a) and (b) is a combination of a maximal level-1 matching (with 5 edges) and a maximal level-3 matching (with 5 edges), which is showed in table 2. The calculated similarity is \((5 + 5*0.5)/13 = 7.5/13\).

<table>
<thead>
<tr>
<th>Level-1 matching ID</th>
<th>o</th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>a</th>
<th>b</th>
<th>Level-3 matching ID</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-1-4</td>
<td>o1</td>
<td>x2</td>
<td>y1</td>
<td>z1</td>
<td>a1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level-1-5</td>
<td>o1</td>
<td>x2</td>
<td>y2</td>
<td></td>
<td>a1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level-1-6</td>
<td>o1</td>
<td>x2</td>
<td>y2</td>
<td></td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level-1-7</td>
<td>o1</td>
<td>x2</td>
<td>y3</td>
<td>z2</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level-1-8</td>
<td>o2</td>
<td></td>
<td>y3</td>
<td>z2</td>
<td>a2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: AND- and OR-matchings for the query graph over the process graph in figure 17
Kanze et. al. proposed several algorithms for computing level-1 matchings under different matching semantics, one of them (denoted as NavOR algorithm) deals with OR-matchings. NavOR algorithm can also be applied for a query subgraph rooted at a node $q_i$ at level $i$ to find level-$i$ matchings for $q_i$. For example, we apply NavOR algorithm two times (one time for query subgraph rooted at $y$, and the other time for query subgraph rooted at $c$) to find all level-3 OR-matchings for the query graph in figure 17. A possible way to discover maximal matchings is to run NavOR algorithm level by level to find possible level-$i$ OR-assignments (level-$i$ assignments for short), and examine combinations of these assignments. We begin to find all level-1 assignments, and store the maximal level-1 assignments in a collection. Then we iterate through the level-2 query nodes, if we find a maximal level-2 assignment combined with a maximal level-1 assignment is larger than the maximal level-1 assignment, we replace the maximal level-1 assignment by the new discovered one. We continue with the computation until all query nodes are processed. The disadvantage of this method is inefficient due to finding level-$i$ assignments by repeatedly invoking NavOR algorithm. After computation of level-$x$ OR-assignments, some OR-level-$y$ assignments ($y > x$) may have been computed. They are computed again, when NavOR algorithm is invoked for query subgraphs rooted at a query node at level $y$. We developed a new data structure (see section 3.4.2) and a new algorithm (see section 3.4.3) to compute maximal level-$i$ OR-assignments efficiently.

### 3.4.2 Data Structure

We call the new data structure solution streams. Like previous mentioned stacks, streams and pools, a solution stream $R_q$ is associated with each query node $q$. Each solution stream $R_q$ consists of following data:

- A stream list, which is a list of stream elements with the type StreamItem. StreamItem is a data structure consisting of: (a) a process node $p$ (called matching process node) that matches predicate of query node $q$; (b) the number of edges of the maximal matching between the process subgraph rooted at $p$ and the query subgraph rooted at $q$ (called maximal matching size for $p$ and $q$ for short); (c) a map which maps each query child node with references to the corresponding child stream elements (called children reference map).
- Size of the maximal level-$i$ OR-matchings for query subgraph rooted at $q$ (called maximal matching size for $q$).

Figure 18 illustrates some initialized solution streams for the query graph and process graph in figure 17. We can access attributes of solution stream $R_q$ by invoke $R_q$.streamList, $R_q$.maxMatchSize. A stream
element at position $i$ of the stream list can be accessed by $R_q$.streamList[$i$]. Attributes of a stream element $se$ can be accessed by invoke $se$.processNode, $se$.maxMatchSize, $se$.childrenRefMap.

Before we run the inexact matching algorithm, we initialize the solution streams with procedure InitSolutionStreams presented in Listing 6. In our algorithm, following notation is used. Let qNodes[] denote an array of query nodes sorted according to level order (see definition 3.5). Note that level order implies topological order for a DAG. The reason is that if node $q_a$ is ancestor of node $q_b$, there is a path from $q_a$ to $q_b$, the longest root-to-$q_b$ path must be longer than the longest root-to-$q_a$ path, so $q_a$ is before $q_b$ in the Array of nodes sorted by level order. If a query node is at position $i$ of qNodes[], then it is accessed by qNodes[$i$] (denoted $q_i$). For a graph object such as query graph $Q$, we denote $|Q|$ the size of $Q$, which is measured by number of edges in $Q$. Besides that, we make use of following functions for a query graph.

Function isAncestor($q_a$, $q_b$) determines whether query node $q_a$ is ancestor of query node $q_b$ by using the pre-computed transitive closure of the query graph. Function isJoinNode($q$) checks whether $q$ is a join node (see definition 2.13). Function getNodesAtLevel($i$) returns query nodes at level $i$. Function isMatch($q$, $p$) determines whether process node $p$ satisfies query node predicate of $q$. Functions described in section 3.2.2 such as isLeaf($q$), subtreeNodes($q$), can also be used.

**Procedure initSolutionStreams($Q$, $P$)**

Input: $Q$: a query graph  
$P$: a process graph  

Result: Initialize solution streams for the query graph

// Phase 1
01 let qNodes[] = [$q_0$, $q_1$, $q_2$, ..., $q_k$] be an array on all query nodes sorted by level order  
02 for $i$ = 0 to $k$  
03 create a new solution stream $R_{q_i}$  
04 if (isLeaf($q_i$))  
05 // Subgraph of a leaf node has no edge, therefore, its matching size is 0  
06 $R_{q_i}$.maxMatchSize = 0  
07 else  
08 // $R_{q_i}$.maxMatchSize = -1 means that the size of the maximal matching is unknown  
09 $R_{q_i}$.maxMatchSize = -1  
10 for each $p$ in $V_P$  
11 if (isMatch($q_i$, $p$)) // $p$ satisfies $q_i$’s predicate  
12 create a new stream element $se$, where $se$.processNode = $p$  
13 if (isLeaf($q_i$))  
14 $se$.maxMatchSize = 0

65
Procedure `initSolutionStreams` works as follows. For each query node, we create and initialize a new solution stream, and iterate through each process node, for each matching process node, we create a new stream element and add it to the stream list (lines 1-15). In the second phase, we iterate through all stream elements. For each stream element `se_q`, we determine ancestor-descendant relationships between process node of `se_q` and process node of each child stream element (if any), and add a reference accordingly (lines 16-19). The worst-case runtime complexity of procedure `initSolutionStreams` is $O(|V_Q||V_P| + |V_Q|^2c^2)$ where $|V_Q|$ and $|V_P|$ denote number of nodes of the query graph and the process graph, $c$ is average number of children of query nodes, $b$ is average size of stream lists. Figure 18 shows some initialized solution streams for the query graph and process graph in figure 17.
Besides solution streams, we use also other data structures to store maximal level-i assignments for all query nodes. We associate each query node \( q \) with a set denoted \( \text{MaxAssignSet} \), which contains maximal assignments for the query subgraph rooted at \( q \). We store and access maximal level-i assignments for \( q \) with \( \text{MaxAssignSet}[q] \). We also associate each process node \( p \) stored in each solution stream of \( q \) a temporary set denoted \( \text{TempMaxAssignSet} \), which contains maximal assignments whose start assign is \([q/p]\). As we will see in section 3.4.3, computation of level-\( l \) maximal assignments for a query subgraph rooted at \( q \) is based on its children assignments. In order to reduce the space complexity, the temporary sets for nodes at certain level \((l_{\text{max-child}} + 1)\) are deleted after we reach a new level \( l \) during the computation of maximal level-\( l \) assignments, where \( l_{\text{max-child}} \) is level of a child node of a level-\( l \) node, which is maximal among all children nodes of level-\( l \) nodes. Because they are not used any more for the computation of maximal level-\( x \) \((x <= l)\) assignments.

### 3.4.3 Computing Maximal Matchings

The basic idea of our new inexact matching algorithm is to find possible level-i OR-assignments for all query nodes, and examine combinations of these assignments. A maximal matching (see definition 2.17) is a level-i assignment or a combination of level-i OR-assignments, which provides the maximal similarity. The discovered maximal matchings are returned.

Our inexact matching algorithm is presented in Listing 7. Algorithm EvalQueryInexactMatch run through all input process graphs. For each process graph \( P_i \), if \( P_i \) is a rooted DAG, we compute maximal matchings between \( P_i \) and \( Q \) that consists of three phases: (1) initialize solution streams as described in section 3.4.2. (2) Compute all maximal level-i assignments for all query nodes. In the second phase and third phase, we differentiate between query tree case and query DAG case. We use procedure computeMaxAssignForTreeNodes presented in Listing 8 for the case that query is a tree, and use procedure computeMaxAssignForDAGNodes presented in Listing 9 for the case that query is a rooted DAG. (3) Find combinations of these level-i assignments that form the maximal matchings by using function maxMatchsOfQueryTree presented in Listing 10 for the query tree case and function maxMatchsOfQueryDAG presented in Listing 11 for the query DAG case.

Some notations are used in the algorithm. Let \( \mu \) denote an assignment, which assigns a query subgraph to a substructure of process graph, \( M = \{\mu_1, \mu_2, \ldots, \mu_n\} \) denote a matching, which contains at least one assignment, \( S = \{M_1, M_2, \ldots, M_k\} \) denote a set of matchings. During the computation, some operations are associated with an assignment \( \mu \). If is \( \mu \) an assignment, \( v \) is a query node that is not yet bound by \( \mu \), and \( p \) is a process node, then \( \mu \oplus [v/p] \) is the assignment that binds \( v \) to \( p \) in addition to the process nodes already bound by \( \mu \) while \( \mu \oplus [v/\bot] \) is the assignment that binds \( v \) to null. If \( \mu_1 \) and \( \mu_2 \) are two different
assignments, $\mu_1 \oplus \mu_2$ combines both assignments to form a new assignment, while $\mu_1 \ominus \mu_2$ removes part of assigns in $\mu_1$, whose query nodes to be assigned appear in $\mu_2$. A similar operation $\mu_1 \ominus \text{QueryNodesSet}$ removes part of assigns in $\mu_1$, whose query nodes to be assigned appear in QueryNodesSet. For example, Level-1-1 assignment showed in table 2 is formed by $\mu_{\text{level-1-1-old}} \ominus \mu_{\text{level-3-1}}$ (or $\mu_{\text{level-1-1-old}} \ominus \text{subgraphNodes(c)}$), where $\mu_{\text{level-1-1-old}}$ is the Level-1-1 OR-assignment in table 1, and $\mu_{\text{level-3-1}}$ is the Level-3-1 assignment in table 2.

Algorithm EvalQueryInexactMatch($Q, P$)

Input: $Q$: a query graph
$PSet$: a set of process graphs $PSet = \{P_1, P_2, ..., P_n\}$
Output: a sorted list of process graphs from $PSet$ with additional matching information such as matching similarity. The list is sorted by matching similarity in descending order

01 let resultList be an empty list used to store matching results information
02 for each $P_i$ in $PSet$
03     if ($P_i$ is a rooted DAG)
// Phase 1
04       initSolutionStreams($Q, P_i$)
05     if ($Q$ is a tree)
// Phase 2
06       computeMaxAssignForTreeNodes($Q, P_i$)
// Phase 3
07       $S_{\text{max}} = \text{maxMatchsOfQueryTree}(Q, P_i)$
08     else if ($Q$ is a rooted DAG)
// Phase 2
09       computeMaxAssignForDAGNodes($Q, P_i$)
// Phase 3
10       $S_{\text{max}} = \text{maxMatchsOfQueryDAG}(Q, P_i)$
11     Calculate similarity $s$ for a maximal matching $M_{\text{max}} \in S_{\text{max}}$ by using connected component based similarity metric
// $s > 1/|Q|$ means that there is at least one matching edge in $M_{\text{max}}$. Alternatively, we can use a // customized similarity threshold instead of $1/|Q|$.
12     if ($s > 1/|Q|$)
13       Create a Matching Result $r_i$, which contains $P_i, S_{\text{max}},$ similarity $s$
14       resultList.add($r_i$)
15 sort resultList by similarity in descending order
16 display results stored in resultList
Phase 2

In phase 2, we compute maximal level-i assignments for all query nodes in descending level order. Assume that the maximal level of a query tree is \(l\). We firstly iterate through query nodes at \(l\), which are leaf nodes. For a query leaf node \(q_l\) and its matching process nodes \(P_{l-match} = \{p_{l1}, ..., p_{ln}\}\), assignments for \(q_l\) are \(\{[q_l / p_{l1}], ..., [q_l / p_{ln}]\}\), all of them are maximal assignments for \(q_l\) with size 0. If \(P_{l-match}\) is empty, assignment for \(q_l\) is \(\{[q_l / \bot]\}\). These assignments are stored in \(MaxAssignSet\) and \(TempMaxAssignSet\). Now we assume that maximal level-\(x\) assignments (\(y < x <= l\)) have been computed, we begin to compute maximal level-\(y\) assignments for a query node \(q_y\) at level \(y\), whose matching process nodes stored in \(q_y\)'s solution stream are \(P_{d-match} = \{p_{d1}, ..., p_{dn}\}\). We iterate through each process node \(p_{dj}\) in \(P_{d-match}\) to compute maximal assignments whose root assign is \([q_y / p_{dj}]\). Among those computed assignments we store the assignments with maximal size in \(MaxAssignSet[q_y]\) as maximal assignments for query subgraph rooted at \(q_y\). Assume that \(p_{d1}, ..., p_{dj-1}\) are processed. We begin to compute maximal assignments whose root assign is \([q_y / p_{dj}]\), which can be calculated through combining assignments of its children. We differentiate between tree case and DAG case.

We first examine the tree case. Two query subtrees rooted at two arbitrary nodes at the same level are not overlapping due to characteristics of tree. For a query node \(q\) at level \(l_x\) in a query tree, all children nodes of \(q\) are at the same level, namely, at level \(l_x - 1\). That is query subtrees rooted at children nodes of \(q\) are not overlapping, and the corresponding assignments can simply be combined. The situation is illustrated in figure 19. Assume that \(q_y\) has two children nodes \(q_{dc1}, q_{dc2}\). In solution stream of \(q_{dc1}\), there is no stream element referenced by stream element of \(p_{dj}\) (denoted \(se_{pdj}\)) in \(q_y\)'s solution stream, namely, \(se_{pdj}.childrenRefMap.get(q_{dc1}).isEmpty\) is true. In solution stream of \(q_{dc2}\), some stream elements are referenced by stream element of \(se_{pdj}\), among them \(se_{dc2}.maxMatchSize\) is maximal, the corresponding assignments are stored in \(TempMaxAssignSet[q_{dc2}, p_{c2m}]\), where \(p_{c2m} = se_{c2m}.processNode\). The new assignments with root assign \([q_y / p_{dj}]\) are calculated as follows: \(\text{TempMaxAssignSet}[q_y, p_{dj}] = \{[q_y / p_{dj}] \oplus \mu_{dc1}, \mu_{dc2} | \mu_{dc1}, \mu_{dc2} \in \text{TempMaxAssignSet}[q_{dc2}, p_{c2m}]\}\), where \(\mu_{dc2}\) is an assignment, which assigns all subtree nodes of \(q_{dc2}\) to \(\bot\). The maximal matching size is \(se_{pdj}.maxMatchSize = se_{c2m}.maxMatchSize + 1\), since an additional query edge \((q_y, q_{dc2})\) is matched. Procedure computeMaxAssignForTreeNodes presented in Listing 8 works follow this principle.
We now examine the DAG case. If a query node $q$ has more than one child node in a DAG, query subgraphs rooted at children nodes of $q$ may be overlapping. Assume that $q_d$ has three children nodes $q_{dc1}$, $q_{dc2}$, $q_{dc3}$. Query graphs rooted at $q_{dc2}$ and $q_{dc3}$ are overlapping, others are not. The situation is illustrated in figure 20. In solution stream of $q_{dc1}$, there is no stream element referenced by the stream element (denoted $s_{pdj}$) in $q_d$’s solution stream. In solution stream of $q_{dc2}$ ($q_{dc3}$), some stream elements are referenced by the stream element $s_{pdj}$, among them $s_{e2m}$.maxMatchSize ($s_{e3m}$.maxMatchSize) is maximal, in addition, $s_{e3m}$.maxMatchSize > $s_{e2m}$.maxMatchSize. We sort query children nodes in that order: (1) children nodes, whose stream elements are referenced by $s_{pdj}$, are before children nodes, which has no stream element referenced by $s_{pdj}$. (2) children nodes, whose stream elements are referenced by $s_{pdj}$, are sorted in descending order of their maximal matching size stored in “maxMatchSize” attribute of the referenced stream element. We iterate through $q_{dc3}$, $q_{dc2}$, $q_{dc1}$, and combine assignments in that order. The maximal assignments started with $[q_d, p_{dj}]$ are calculated as follows: $TempMaxAssignSet[q_d, p_{dj}] = \{[q_d / p_{dj}] \oplus \mu_{dc3} \oplus \mu_{dc2-part} \mid \mu_{dc3} \in TempMaxAssignSet[q_{dc3}, p_{c3m}], \mu_{dc2-part} \in \{\mu_{dc2} \ominus (subtreeNodes(q_{dc2}) \cap subtreeNodes(q_{dc3})) \mid \mu_{dc2} \in TempMaxAssignSet[q_{dc2}, p_{c2m}]\} \}$, where $\mu_{dc1}$ is an assignment, which assigns all subtree nodes of $q_{dc1}$ to $\perp$. The only difference to the tree case is $\mu_{dc2-part}$, which is bound after $\mu_{dc3}$. $\mu_{dc3}$ assigns values for query subgraph rooted at $q_{dc3}$, while $\mu_{dc2}$ assigns values for query subgraph rooted at $q_{dc2}$. The both query subgraphs are overlapping. We must decide which values to be assigned for the common subgraph. Because $|\mu_{dc3}| > |\mu_{dc2}|$ and $\mu_{dc3}$ is bind to $[q_d / p_{dj}]$ first, the common part is defined by $\mu_{dc3}$. We combine $\mu_{dc2-part}$ instead of $\mu_{dc2}$ with $[q_d / p_{dj}] \oplus \mu_{dc3}$, where $\mu_{dc2-part}$ removes part of assigns in $\mu_{dc2}$ whose query nodes to be assigned in the common subgraph. The maximal matching size is $s_{pdj}$.maxMatchSize = $s_{e3m}$.maxMatchSize + $|\mu_{dc2} \ominus (subtreeNodes(q_{dc2}) \cap subtreeNodes(q_{dc3}))| + 2$. 

Figure 19: Compute maximal assignments for a subtree rooted at a non-leaf node in query tree
Figure 20: Compute maximal assignments for a subgraph rooted at a non-leaf node in a query DAG

Procedure computeMaxAssignForTreeNodes($Q$, $P$)

Input: $Q$: a query tree  
$P$: a process rooted DAG

Output: maximal matchings and the number of edges of each of these maximal matchings for all stream elements

01 $qNodes[] = [q_0, q_1, q_2, ..., q_k]$ is an array on all query nodes sorted by level order
02 for $i = k$ to 0  // query nodes are processed in bottom-up order (from leaf to root)
03 $l_i = getLevel(q_i)$  // is $l_i$ the level of $q_i$ in the query tree
04 if ($i > 0$) and ($l_i > (getMaxLevel(Q) - 2)$) and (getLevel($q_i$) > getLevel($q_{i-1}$))
05 delete all level-$(l_i - 2)$ assignments in $TempMaxAssignSet$
06 if (isLeaf($q_i$))
07 for each stream element $se_{q_i}$ in $R_{q_i}.streamList$
08 $p_x = se_{q_i}.processNode$
09 $\mu_{px} = [q_i / p_x]$
10 $TempMaxAssignSet[q_i, p_x] = \{\mu_{px}\}$
11 $MaxAssignSet[q_i] = MaxAssignSet[q_i] \cup \{\mu_{px}\}$
12 $se_{q_i}.maxMatchSize = 0$
13 else  // $q_i$ is not a leaf node
14 for each stream element $se_{q_i}$ in $R_{q_i}.streamList$
15 $p_x = se_{q_i}.processNode$
16 $\mu_{px} = [q_i / p_x]$  // the start assign
17 // Count the number of edges of the maximal assignment, whose start assign is $[q_i / p_x]$
18 $maxSizePx = 0$
19 $TempMaxAssignSet[q_i, p_x] = \{\mu_{px}\}$
20 // iterate through children nodes to calculate assignments for $q_i$ through combine

71
for each \( q_c \) in children(\( qi \))

// No references to any stream elements stored in solution stream of \( q_c \)

if \( (se_{qi}.childrenRefMap.get(q_c).isEmty) \)

// Expand each assignment with undefined for each query node in \( q_c \)’s subtree, replace
// the old assignment with the expanded one.

for each in \( \mu_{px} \) in TempMaxAssignSet[\( qi, px \)]

\[
TempMaxAssignSet[\( qi, px \)] = \TempMaxAssignSet[\( qi, px \)] \setminus \{ \mu_{px} \}
\]

for each \( q_{csub} \) in subtreeNodes(\( q_c \))

\[
\mu_{px} = \mu_{px} \oplus [q_{csub} / \perp]
\]

\[
TempMaxAssignSet[\( qi, px \)] = \TempMaxAssignSet[\( qi, px \)] \cup \{ \mu_{px} \}
\]

else

let \( SE_{qc-max} \) be a set of stream elements referenced by \( se_{qi}.childrenRefMap.get(q_c) \), each
of these stream elements has a maximal value \( m \) for attribute “maxMatchSize” among
all referenced stream elements in \( se_{qi}.childrenRefMap.get(q_c) \)

\[
maxSizePx = maxSizePx + m + 1
\]

for each in \( \mu_{px} \) in TempMaxAssignSet[\( qi, px \)]

\[
TempMaxAssignSet[\( qi, px \)] = \TempMaxAssignSet[\( qi, px \)] \setminus \{ \mu_{px} \}
\]

for each \( se_{qc-max} \) in \( SE_{qc-max} \)

\[
p_{qcm} = se_{qc-max}.processNode
\]

for each in \( \mu_{pqcm} \) in TempMaxAssignSet[\( q_o, p_{qcm} \)]

\[
TempMaxAssignSet[\( qi, px \)] = \TempMaxAssignSet[\( qi, px \)] \cup \{ \mu_{px} \ominus \mu_{pqcm} \}
\]

// end of the loop in line 19

// maximal assignments with start assign \( [q_i / p_x] \), where \( p_x = se_{qi}.processNode \)

// have been computed

\[
se_{qi}.maxMatchSize = maxSizePx
\]

// maximal assignments with start assign \( [q_i / p_x] \) is larger than the existing maximal
// assignments for \( qi \)

if \( (R_{qi}.maxMatchSize < maxSizePx) \)

\[
R_{qi}.maxMatchSize = maxSizePx
\]

\[
MaxAssignSet[\( qi \)] = TempMaxAssignSet[\( qi, px \)]
\]

else if \( (R_{qi}.maxMatchSize == maxSizePx) \)

\[
MaxAssignSet[\( qi \)] = MaxAssignSet[\( qi \)] \cup TempMaxAssignSet[\( qi, px \)]
\]

Listing 8: Procedure computeMaxAssignForTreeNodes
In Procedure computeMaxAssignForTreeNodes, we iterate through query nodes in level order (line 2). For each query node $q_i$ at a new level $l_i$ ($l_i > 2$), we delete all level-$(l_i - 2)$ assignments in $TempMaxAssignSet$ (lines 3-5), since they are not used for computation any more. In lines 6-12, if $q_i$ is a query leaf node and its matching process nodes stored in solution stream are $\{p_{x1}, \ldots, p_{xn}\}$, maximal assignments for $q_i$ are $\{[q_i / p_{x1}], \ldots, [q_i / p_{xn}]\}$, otherwise, if it has no matching process node, assignment for $q_i$ is $\{[q_i / \bot]\}$. These assignments are stored in $MaxAssignSet$ and $TempMaxAssignSet$. Now we examine the case that $q_i$ is not a query leaf node (lines 13-34). Assume that $q_i$ is at level $l$. Maximal level-$x$ assignments ($l < x$) are computed, since maximal matchings for query nodes are computed in that order. We begin to compute maximal level-$l$ assignments whose root assign is $[q_i / p_x]$ (line 16), which can be calculated through combining assignments of its children. For each child node $q_c$ of $q_i$, we differentiate two cases. If no stream element in solution stream of $q_c$ is referenced by stream element of $p_x$ (denoted $seq_i$) in $q_i$’s solution stream (line 20), we expand each assignment $\mu_{px}$ stored in $TempMaxAssignSet[q_i, px]$ to $\mu_{px} \oplus \mu_c$, where $\mu_c$ is an assignment, which assigns all subtree nodes of $q_c$ to $\bot$ (lines 21-25). Otherwise, we iterate through maximal assignments, where each of them (denoted $\mu_{pcm}$) has a start assign $[q_c / p_{cm}]$, and stream element of $p_{cm}$ is referenced by stream element of $p_x$ in $q_i$’s solution stream, we expand $\mu_{px}$ stored in $TempMaxAssignSet[q_i, px]$ to $\mu_{px} \oplus \mu_{pcm}$ (lines 27-34). After all children assignments are processed, maximal assignments with start assign $[q_i / p_x]$ have been computed and stored in $TempMaxAssignSet[q_i, px]$. If new computed maximal assignments with start assign $[q_i / p_x]$ are larger than existing maximal assignments for query subtree rooted at $q_i$ (lines 36-38), we replace existing maximal assignments $MaxAssignSet[q_i]$ with them. Otherwise, if they have the same size as existing maximal assignments, we add them to $MaxAssignSet[q_i]$ (lines 39-40). After all stream elements in $q_i$’s solution stream are processed, we get maximal assignments for query subtree rooted at $q_i$ and they are stored in $MaxAssignSet[q_i]$. The worst-case runtime complexity of procedure computeMaxAssignForTreeNodes is $O(|V_Q|b^2c^2m^2)$, where $|V_Q|$ denotes number of nodes of the query graph, $c$ is average number of children of query nodes, $b$ is average size of stream lists, $m$ is average size of $MaxAssignSet[q_i]$.

**Procedure computeMaxAssignForDAGNodes($Q$, $P$)**

Input: $Q$: a rooted DAG pattern query

$P$: a rooted DAG pattern process

Output: maximal matchings and the number of edges of each of these maximal matchings for all stream elements

01 qNodes[] = [$q_0$, $q_1$, $q_2$, ... , $q_k$] is an array on all query nodes sorted by level order
02 for $i = k$ to 0  // query nodes are processed in bottom-up order (from leaf to root)
03     $l_i$ = getLevel($q_i$)  // is $l_i$ the level of $q_i$ in the query tree
04     if (($i > 0$) and ($l_i > (getMaxLevel($Q$) - 2)$) and (getLevel($q_i$) > getLevel($q_{i-1}$)))
05         delete all level-$(l_i - 2)$ assignments in $TempMaxAssignSet$
if (isLeaf(qi))
    for each stream element seqi in Rqi.streamList
        px = seqi.processNode
        μpx = [qi / px]
        TempMaxAssignSet[qo, px] = {μpx}
        MaxAssignSet[qi] = MaxAssignSet[qi] ∪ {μpx}
        seqi.maxMatchSize = 0
else   // qi is not a leaf node
    for each stream element seqi in Rqi.streamList
        px = seqi.processNode
        μpx = [qi / px]  // the start assign
    // Count the number of edges of the maximal assignment, whose start assign is [qi / px]
    maxSizePx = 0
    TempMaxAssignSet[qo, px] = {μpx}
    // store nodes for query subgraphs rooted at processed children nodes of qi
    processedChildrenSubtreeNodes = {}
    let CList be a list of query children nodes of qi, which are sorted by 2 criteria: (1) children
    nodes, whose stream elements are referenced by seqi, are before children nodes, which has no
    stream element referenced by seqi. (2) children nodes, whose stream elements are referenced
    by seqi, are sorted in descending order of their maximal matching size stored in
    “maxMatchSize” attribute of the referenced stream element.
    // iterate through sorted children nodes to calculate assignments started with [qi / px]through
    // combine children assignments
    for each qc in CList
        // CommonNodes are intersection of processed subtree nodes and nodes of subtree rooted at
        // qc
        CommonNodes = (processedChildrenSubtreeNodes ∩ subtreeNodes(qc))
        processedChildrenSubtreeNodes.add(subtreeNodes(qc))
        // No references to any stream elements stored in solution stream of qc
        if (seqi.childrenRefMap.get(qc).isEmpty)
            // Expand each assignment with undefined for each query node in qc’s substree
            for each μpx in TempMaxAssignSet[qo, px]
                TempMaxAssignSet[qo, px] = TempMaxAssignSet[qo, px] \ {μpx}
            if (CommonNodes) is empty
            for each qsub in subtreeNodes(qc)
                μpx = μpx ⊕ [qsub / ⊥]
        else
for each $q_{cub}$ in (subtreeNodes($q_c$) \ $CommonNodes$)

$$\mu_{px} = \mu_{px} \oplus [q_{cub} / ⊥]$$

$TempMaxAssignSet[q_i, p_x] = TempMaxAssignSet[q_i, p_x] \cup \{ \mu_{px} \}$

else

let $SE_{qc-max}$ be a set of stream elements referenced by $seq_{i}.childrenRefMap.get(q_c)$, each of these stream elements has a maximal value $m$ for attribute “maxMatchSize” among all stream elements referenced by $seq_{i}.childrenRefMap.get(q_c)$

for each $\mu_{px}$ in $TempMaxAssignSet[q_i, p_x]$

if ($CommonNodes$ is empty)

$TempMaxAssignSet[q_i, p_x] = TempMaxAssignSet[q_i, p_x] \setminus \{ \mu_{px} \}$

for each $se_{qc-max}$ in $SE_{qc-max}$

$$p_{pcm} = se_{qc-max}.processNode$$

for each $\mu_{pcm}$ in $TempMaxAssignSet[q_i, p_{pcm}]$

$$\mu_{pcm1} = \mu_{pcm} \ominus CommonNodes$$

if ($maxPartialAssignSize < |\mu_{pcm1}|$)

$maxPartialAssignSize = |\mu_{pcm1}|$

$TempMaxAssignSet[q_i, p_x] = \{ \mu_{px} \ominus \mu_{pcm1} \}$

else if ($maxPartialAssignSize == |\mu_{pcm1}|$)

$TempMaxAssignSet[q_i, p_x] = TempMaxAssignSet[q_i, p_x] \cup \{ \mu_{px} \ominus \mu_{pcm1} \}$

// end of the loop in line 36

if ($CommonNodes$ is empty)

$maxSizePx = maxSizePx + m + 1$

else

$maxSizePx = maxSizePx + maxPartialAssignSize + 1$

// end of the loop in line 21

$maxSizePx = maxSizePx$

// maximal assignments with start assign $[q_i / p_x]$, where $p_x = se_{q_i}.processNode$ have been computed

$se_{q_i}.maxMatchSize = maxSizePx$

// the new computed maximal assignments with start assign $[q_i / p_x]$ is larger than the existing
// maximal assignments for \( q_i \)
60    if (\( R_{q_i}.\text{maxMatchSize} < \text{maxSizePx} \))
61        \( R_{q_i}.\text{maxMatchSize} = \text{maxSizePx} \)
62        \( \text{MaxAssignSet}[q_i] = \text{TempMaxAssignSet}[q_o, p_x] \)
63    else if (\( R_{q_i}.\text{maxMatchSize} == \text{maxSizePx} \))
64        \( \text{MaxAssignSet}[q_i] = \text{MaxAssignSet}[q_i] \cup \text{TempMaxAssignSet}[q_o, p_x] \)

Listing 9: Procedure computeMaxAssignForDAGNodes

Procedure computeMaxAssignForDAGNodes is similar to procedure computeMaxAssignForTreeNodes. The basic idea of both procedures is the same. Namely, iterate through query nodes level by level in bottom up direction. For each query node \( q_i \), we calculate maximal assignments for query subtree rooted at \( q_i \) based on maximal assignments for query subtrees rooted at children nodes of \( q_i \). In the following, we focus on the code specific for the DAG characteristic. Line numbers of code, which are different from the procedure computeMaxAssignForTreeNodes, are formatted bold. We examine the case that \( q_i \) is not a query leaf node. We begin to compute maximal assignments whose root assign is \( [q_i / p_x] \), where \( p_x \) is a matching process node stored in solution stream of \( q_i \). We first try to combine \( [q_i / p_x] \) with child assignments for a query subgraph rooted at a child node, which have the largest size among all children assignments whose stream element is referenced by stream element with \( p_x \), then combine with such child assignments with second large size and so on. Children nodes of \( q_i \) are processed in that order (lines 20-21). For each query node \( q_o \), we calculate intersection of \( \text{processedChildrenSubtreeNodes} \) and \( \text{subgraphNodes}(q_c) \) and store them in \( \text{CommonNodes} \) (line 22). If \( \text{CommonNodes} \) is empty, that means query subgraphs rooted at \( q_i \) is not overlapping with any previously processed subgraphs rooted at \( q_i \)'s children, then we expand assignments same as the tree case (lines 24-29 and 36-42). Otherwise, we expand assignments only for those nodes in subtreeNodes\( (q_o) \) which are not included in \( \text{CommonNodes} \) (lines 30-32 and 43-54). The worst-case runtime complexity of procedure computeMaxAssignForDAGNodes is \( O(|V_{\text{c}}|(|V_{\text{c}}| + b^2c^2)) \), where \( |V_{\text{c}}| \) denotes number of nodes of the query graph, \( c \) is average number of children of query nodes, \( b \) is average size of stream lists, and \( m \) is average number of assignments of \( \text{TempMaxAssignSet}[q_o, p_x] \), where \( m \) is bound by \( |V_{\text{c}}|^b \).

Phase 3
In phase 3, we find maximal matchings between the query graph and the process graph as combinations of maximal level-1 assignments computed in phase 2. The two procedures presented in Listing 10 and Listing 11 process all maximal level-1 assignments in a top-down fashion. A set \( S_{\text{max}} \) is used to store discovered maximal matchings so far. We begin with query root (level-1) and store all maximal level-1 matchings in \( S_{\text{max}} \). Assume that we process a query node \( q_d \) at level \( y \) in the query graph, all maximal level-1 assignments for \( q_d \) are stored in \( \text{MaxAssignSet}[q_d] \). We combine each in \( \mu_d \) in \( \text{MaxAssignSet}[q_d] \) with \( M_{\text{max}} \) in \( S_{\text{max}} \). We
differentiate between tree case and rooted DAG case by the combination of the assignments as described as follows. If a matching $M_{\text{max-new}}$, which is a combination of $\mu_d$ and $M_{\text{max}}$, is larger than $M_{\text{max}}$, then we replace maximal matchings in $S_{\text{max}}$ with $M_{\text{max-new}}$.

For a query tree, if $q_d$ is assigned by $\mu_m \in M_{\text{max}}$, query subtree rooted at $q_d$ is a subsubtree of the query subtree assigned by $\mu_m$, otherwise, there is no common nodes or edges between them. In other words, for each $M_{\text{max}} \in S_{\text{max}}$, there is exactly one assignment (denoted $\mu_m$) includes query subtree rooted at $q_d$ and this assignment contains parent node of $q_d$. Thus, for $M_{\text{max}}$ we need only to test combinations between $\mu_m$ and assignments in $\text{MaxAssignSet}[q_d]$. We prove that through induction. If $q_d$ is a level-2 query node, each $M_{\text{max}} \in S_{\text{max}}$ contains only one maximal level-1 assignment (denoted $\mu_m$). The first query node assigned by $\mu_m$ is query root, which is parent of $q_d$. $\mu_m$ assigns the entire query tree, which includes the subtree rooted at $q_d$. Now we assume that $q_d$ is a level $y$ ($y>2$) query node, and $q_{da}$ is ancestor of $q_d$, and $q_d$ has $n$ children nodes $q_{dc1}, \ldots, q_{dcn}$. This is showed in figure 21. In a maximal matching $M_{\text{max}} \in S_{\text{max}}$, an assignment $\mu_d \in \text{MaxAssignSet}[q_d]$ is either combined with an assignment $\mu_{da} \in M_{\text{max}}$, which includes parent node $q_d$ and query subtree rooted at $q_d$, or not. In the first case, $M_{\text{max}} = (M_{\text{max}} \setminus \mu_{da}) \cup \{\mu_d, (\mu_{da} \Theta \mu_d)\}$, and $\mu_d$ is the only assignment which assigns query subtree rooted at $q_d$, we continue to process siblings nodes of $q_d$ at level $y$, their subtrees are not overlapping with the subtree rooted at $q_d$, therefore, the corresponding level-$y$ assignments are not combined with $\mu_d$. The same happens if we process query nodes at level $y+1$ except children nodes of $\mu_d$. Thus, $\mu_d$ is the only assignment in $M_{\text{max}}$ that assigns query subtree rooted at $q_d$, before we process children nodes of $q_d$. Assume we processed children nodes $q_{dc2}, \ldots, q_{dc(i-1)}$ and $\mu_{d-new}$ is an assignment in $M_{\text{max}}$ after $\mu_d$ combined with corresponding children assignments. $\mu_{d-new}$ is the only assignment in $M_{\text{max}}$ that assigns query subtree rooted at $q_{dc(i-1)}$, because subtrees rooted at $q_{dc2}, \ldots, q_{dc(i-1)}$ are not overlapping with subtree rooted at $q_{dc(i-1)}$. In the second case, $\mu_{da}$ is the only assignment in $M_{\text{max}}$ that includes query subtree rooted at $q_d$, and the rest is similar to the first case.
Now we examine the rooted DAG case. The corresponding function maxMatchsOfQueryDAG is presented in Listing 11. If we process query nodes level wise, and test combinations of their maximal level-i assignments, for each $\mu_{di}$ which is a maximal level-y assignment for a subgraph rooted at a level-y query node $q_{di}$, it may be overlapping with any assignments stored in $M_{max} \in S_{max}$. Because two subgraphs rooted at two arbitrary query nodes may be overlapping in a rooted DAG query. If we use the combination strategy like in the tree case, it is expensive to test combinations. We adopt a new combination strategy for the rooted DAG case. We sort query nodes in descending order of the size of the maximal level-i assignment stored in “maxMatchSize” attribute of each solution stream (line 1). Assume $R_{q0}.maxMatchSize$ with query node $q_0$ is maximal among all query nodes except query root. We test combinations of all maximal level-1 assignments with $\mu_{q0}$ in $MaxAssignSet[q_0]$, and store maximal matchings in $S_{max}$ (lines 2-13). We iterate through the rest query nodes. For each query node $q_i$ and each $M_{max} \in S_{max}$, we find an assignment $\mu_{m}$ in $M_{max}$ that includes $q_i$, and try to combine $\mu_{m}$ with $\mu_{qi} \in MaxAssignSet[q_i]$ (lines 14-24). The worst-case runtime complexity of procedure $\text{maxMatchsOfQueryDAG}$ is $O(|V_Q|^2m_1m_2)$, where $|V_Q|$ denotes number of nodes of the query graph, $m_1$ is size of $S_{max}$, $m_2$ is average number of assignments of $MaxAssignSet[q_i]$. $m_1$ and $m_2$ are bound by $|V_Q|^b$, where $b$ is average size of stream lists. The worst case runtime complexity of our new inexact matching algorithm for rooted DAG is $O(|V_Q|((|V_P| + cb^2) + (|V_Q| + b^2cm_2) + |V_Q|m_1m_2))$, where $|V_P|$ denotes number of nodes of the process graph, $c$ is average number of children of query nodes, $m$ is average number of assignments of $TempMaxAssignSet[q_i, p_i]$, where $m$ is bound by $|V_Q|^b$. 

Figure 21: Processing of combination of subtree assignments
Function `maxMatchsOfQueryTree(Q, P)`

Input:  
- `Q`: a query tree
- `P`: a process rooted DAG

Output: maximal matchings between the given query tree and process DAG

01 `qNodes[] = [q_0, q_1, q_2, ..., q_k]` is an array on all query nodes sorted by level order

// Store all maximal level-1 assignments in `S_max`, which is a set used to store maximal matchings
02 `S_max = MaxAssignSet[q_0]`

03 for `i = 1` to `k`  // iterate through query tree nodes from level 2
04 for each `M_max` in `S_max`
05 for each in `μ_m` in `M_max`

// If parent node of `q_i` is one of the query nodes assigned by `μ_m`, then we try to combine `μ_m` with
// maximal assignments for `q_i` stored in `MaxAssignSet[q_i]`.
06 if (parent(`q_i`) ∈ getQueryNodes(`μ_m`))
07 let `μ_m1` be part of `μ_m` that includes `subtreeNodes(q_i)`
08 `μ_m2 = μ_m \ O` `subtreeNodes(q_i)`

// Only if `μ_m1` is not the maximal assignment among all assignments rooted at `q_i`, it is
// possible that a combination of `μ_m` and a maximal assignment rooted at `q_i` forms a larger
// assignment.
09 if (|`μ_m1`| < `R_{q_i}.maxMatchSize`)
   // determine whether `μ_m` is maximal among all assignments in `M_max`
10 `isMaxAssign = checkMaxAssign(M_max, μ_m)`
11 for each in `μ_i` in `MaxAssignSet[q_i]`
12 if (((|`μ_i`| >= |`μ_m`|)
    or ((not `isMaxAssign`) and ((|`μ_i`| + |`μ_m2`|) > |`μ_m`|)))
    or (`isMaxAssign` and (c*(|`μ_i`| + |`μ_m2`|) > |`μ_m`|)))  // c is connectivity factor

// we found a new maximal assignment, and replace `S_max` with `{M_max_new}`
13 `M_{max-new} = (M_max\{ μ_m \}) \cup \{μ_i, μ_m2\}`
14 `S_max = {M_{max-new}}`

// end of the loop in line 6
15 return `S_max`

Listing 10: Function `maxMatchsOfQueryTree`

Function `maxMatchsOfQueryTree` accepts an input query tree and an input process rooted DAG, it returns all maximal matchings between them. A maximal matching may consists of several assignments. Each assignment assigns a query subtree to matching subtree in the process graph. We process query nodes level by level in top down order. In line 2, we store all maximal level-1 assignments in `S_max`, which is a set used
to store maximal matchings of query graph and process graph. We iterate through the sorted query nodes from level 2 (line 3). For each assignment $\mu_m$ in each computed maximal matchings stored in $S_{max}$ (lines 4-5), we test whether query nodes assigned by $\mu_m$ are overlapping with query subtree rooted at $q_i$ (line 6). Only if they are overlapping, we test whether a combination of $\mu_m$ and $\mu_i$ (maximal level-i assignments for query subtree rooted at $q_i$) forms a larger matching (lines 7-12). If it is a larger matching, we replace $\mu_m \in M_{max}$ by $\mu_i$ and ($\mu_m \emptyset$ subtreeNodes($q_i$)) (lines 13-14). The worst-case runtime complexity of procedure maxMatchsOfQueryTree is $O(|V_Q| (m_1m_2(|V_Q| + m_2))$, where $|V_Q|$ denotes number of nodes of the query graph, $m_1$ is size of $S_{max}$, $m_2$ is average number of assignments of $MaxAssignSet[q_i]$. $m_1$ and $m_2$ are bound by $|V_Q|*b$, where b is average size of stream lists.

**Function maxMatchsOfQueryDAG(Q, P)**

Input: $Q$: a rooted DAG pattern query graph  
$P$: a rooted DAG pattern process graph  

Output: maximal matchings between the given query DAG and process DAG

01 $qNodes[] = [q_0, q_1, q_2, ... , q_k]$ is an array on all query nodes except the query root node, they are sorted in descending order of the size of the maximal level-i assignment stored in “maxMatchSize” attribute of each solution stream.  
// Combine maximal level-i assignments for query subgraph rooted at $q_0$ with maximal level-1 assignments

02 if ($R_{q_0}.maxMatchSize \geq R_r.maxMatchSize$) // $R_r$ is the solution stream for the root node $r$

03 for each $\mu$ in $MaxAssignSet[q_r]$

04 for each $\mu_{q_0}$ in $MaxAssignSet[q_0]$

05 $S_{max} = S_{max} \cup \{\mu_{q_0}, \mu_r \Theta \mu_{q_0}\}$

06 else

07 $S_{max} = MaxAssignSet[q_r]$

08 for each $\mu$ in $MaxAssignSet[q_r]$

09 for each $\mu_{q_0}$ in $MaxAssignSet[q_0]$

10 if ($c^{*}(\mu_r, \Theta \mu_{q_0}) + |\mu_{q_0}| > |M_{max}|$) // $M_{max} \in S_{max}$

11 $S_{max} = \{\mu_{q_0}, \mu_r \Theta \mu_{q_0}\}$

12 else if ($c^{*}(\mu_r, \Theta \mu_{q_0}) + |\mu_{q_0}| == |M_{max}|$)

13 $S_{max} = S_{max} \cup \{\mu_{q_0}, \mu_r \Theta \mu_{q_0}\}$

14 for $i = 1$ to $k$ // iterate through the sorted query graph nodes

15 for each $M_{max}$ in $S_{max}$

16 find an assignment $\mu_m$ in $M_{max}$ that includes $q_i$

17 $commonNodes = getQueryNodes(\mu_m) \cap$ subtreeNodes($q_i$)

18 let $\mu_{m1} = \mu_m \emptyset$ commonNodes
for each in $\mu_{qi}$ in $MaxAssignSet[q_i]$

let $\mu_{qi1}$ be part of $\mu_{qi}$ that includes $commonNodes$

if ($(\mu_m$ is maximal in $M_{max}$ and $(c^*(|\mu_{qi1}| + |\mu_{m1}|) > |\mu_m|)$)

or ($(\mu_m$ is not maximal in $M_{max}$ and $(|\mu_{qi1}| + |\mu_{m1}|) > |\mu_m|))$

$S_{max} = (S_{max} \setminus M_{max}) \cup \{(M_{max} \setminus \mu_m) \cup \{\mu_{qi1}, \mu_{m1}\}\}$

// end of the loop in line 6

return $S_{max}$

Listing 11: Function maxMatchsOfQueryDAG
Chapter 4: Prototype Realization

Chapter 4 deals with the design and implementation of the prototype for querying BPEL business processes. The main part of the prototype is based on the algorithms in chapter 3. We first give an overview of the applied technologies. Then we present the architecture of the prototype and introduce the main components of the prototype.

4.1 Applied Technologies

The prototype is developed under Eclipse 3.3.1 with Java 1.6, and additional plugins such as EMF 2.3.1. We use Hibernate as persistence layer. Data are stored in PostgreSQL database. The prototype can be easily extended to work with many other databases like My-SQL, DB2, Oracle due to abstraction of Hibernate.

Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [EMF] is an Eclipse-based modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF started out as an implementation of the MOF specification, and can be thought of as a highly efficient Java implementation of a core subset of the MOF API. EMF unifies the three important technologies: Java, XML, and UML. An EMF model is the common high-level representation that "glues" models represented in Java, XML, UML together. The model used to represent models in EMF is a MOF-like meta-model (called Ecore). Ecore is itself an EMF model, and thus is its own meta-model, that makes it also a meta-meta-model. Ecore models can be generated by models with annotated Java, UML, XML representation. On the other side, the corresponding Java classes, XML schema, UML diagrams can be generated based on Ecore. Ecore is serialized as XMI (XML Metadata Interchange), which is a standard for serializing metadata. Besides Ecore, which describes the classes to be generated and their names, attributes, and references and runtime support for the models such as change notification, there is a generator model. It provides some additional user-settable information for the code generation, such as where to put the generated code and what prefix to use for the generated factory and package class names.

The following modules can be generated from the EMF model:
• **Data model** includes Java interfaces, implementation classes and factory classes for all the classes in the EMF model. Except data model, there are some additional functionalities: (1) Methods for serialization from the generated classes and files to XML- or XMI- files and vice versa. (2) A reflective API for manipulating EMF objects (for example, read and write the attributes or define a new class in EMF model in run time). (3) The generated classes are allowed to register listeners for the change notification.

• **Adapter** builds on the generated data model. A set of adapt classes are generated that provide a structured view and perform command-based editing of the model objects

• A basic **editor** displays and edits (that is, copy, paste, drag-and-drop, and so on) instances of the model using standard JFace viewers and a property sheet.

**Hibernate**

Hibernate [HIBER] is an open source object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Many applications involve a lot of efforts to store objects in the database or get data from the database as objects. Writing code to perform these tasks requires complex SQL queries and is tedious and error-prone. Object-relational mapping (ORM) is a layer between relational database and object oriented application. Hibernate not only takes care of the mapping from Java classes to database tables, but also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate's primary feature is mapping from Java classes to database tables (and from Java data types to SQL data types). What you need to do is create an XML mapping file that describe the classes to be stored in a database, and how they relate to the tables and columns in that database. In addition, a hibernate.properties file is used for Hibernate’s configuration. The configuration file contain all necessary information to accesses the database, such as database address, user name, password. At runtime, Hibernate reads the mapping file and dynamically builds Java classes to manage the translation between the database and Java worlds.

Hibernate provides data query and retrieval facilities. A database query can be written in Hibernate’s own object-oriented query language (HQL) or SQL. HQL is a query language similar to the stand SQL, but it can use class name and attribute name in query instead of table name and table column name. The advantage of using HQL instead of SQL is that SQL varies as different databases are used, but HQL remain the same, no matter which database are used. This improves reusability of code. Hibernate generates the SQL calls and relieves the developer from manual result set handling and object conversion, keeping the application portable to all SQL databases, with database portability delivered at very little performance overhead. There is a simple, intuitive API in Hibernate to perform queries against the objects.
represented by the database. Beside that, Hibernate supports different object oriented concept such as polymorphismus, composition and inheritance.

**Teneo**

Teneo is a database persistency solution for EMF using Hibernate or JPOX/JDO 2.0. It supports automatic creation of EMF to Relational Mappings and the related database schemas. The solution also contains a runtime layer to support specific EMF features. EMF Objects can be stored and retrieved using EMF resource approach or HQL queries. EMF objects can be persisted in most major relational databases.

The EMF-Hibernate integration is a layer over the Hibernate product, and it consists of two main parts:

1. **Mapping**: the mapping layer is responsible for the automatic mapping of an EMF model to an in-memory Hibernate mapping. It is possible to programmatically generate a Hibernate mapping file from an EMF model. Hibernate mapping options such as inheritance mapping can be specified to influence the generated mapping file.

2. **Runtime**: the runtime layer takes care of handling specific EMF features such as lazy loading of ELists, setting EFeatures from the database, instantiating EObjects using the EClass name, etc. The main visible component of the runtime layer is the HbDataStore. The HbDataStore controls a SessionFactory and a set of EPackages, which are persisted using the sessions of this session factory. When using a Hibernate session, behind the scenes Teneo takes care of instantiating EMF objects and setting/getting EFeatures from the database. Teneo also handles lazy loading of ELists.

**JGrapht**

JGraphT is a free Java graph library that supports different graph types (such as (un)directed graphs, graphs with (un)weighted or (un)labelled edges) and a number of graph algorithms. The complete source code of JGraphT is included under the terms of the GNU Lesser General Public License. JGraphT is designed to be simple and type-safe (via Java generics). For example, graph vertices can be of any objects. It is possible to visualize graphs created with JGraphT by using the JGraph library.

### 4.2 System Architecture and Implementation

The System consists of four components: (1) the database component deals with store BPEL file in a PostgreSQL database. (2) The graph build component transforms BPEL business processes stored in database to a set of rooted DAGs. In this prototype, the transformation is performed in runtime, however,
in order to achieve a better performance, it is recommended to do this step in pre-processing time and store the generated process graphs in a graph table. The graph table should keep “insync” with the process tables, by update of process tables is the graph table also updated. (3) The query evaluation component evaluates an input query graph under exact matching semantic or inexact matching semantic depends on the input search type. It iterates through all BPEL graphs. For each BPEL graph, an appropriate graph matching algorithm is selected depend on search type, BPEL graph type and query graph type. (4) The GUI component is used to input query and display matching results. It is developed in this prototype due to the time limitation. We use the database component and the main part of the graph build component developed as a part of a BPEL fragment-mining prototype for another diploma thesis [Ger08]. Therefore, we give only a brief overview of both components. The query evaluation component is the main work of this thesis. We introduce it in section 4.2.3 in detail.

---

**Figure 22: System Architecture**

---
4.2.1 Database Component

The database component deals with load and store BPEL-processes in a relational database.

![Diagram of BPEL file, EMF, and Hibernate](image)

**Figure 23: Store a BPEL process in a relational database**

The modeling basis is XML Schema for BPEL 2.0. With the EMF framework, we create an EMF Ecore model from an XML Schema and then generate a simple model editor for it. The basic mapping rules from XML Schema to Ecore are: (1) A schema maps to an EPackage; (2) A complex type definition maps to an EClass; (3) A simple type definition maps to an EDataType; (4) An attribute declaration or a nested element declaration maps to an EAttribute or EReference, depending on its type; (5) An EClass called DocumentRoot is created to hold any top-level element or attribute declarations. The generated Ecore is the meta-model of the BPEL-model. Based on Ecore model for BPEL a set of java classes are generated, these java classes are divided in three package as described in section 4.1. All classes in the BPEL model are derived from the class “ExtensibleElements”.

A specific mapping file named hibernate.hbm.xml is generated to map the EMF model to database schema, in doing so it maps each class to a table and each attribute to a column of the table. For each association generate Hibernate a corresponding foreign key. The mapping file is also generated from EMF model. We set the following persistence options to control the mapping and runtime behaviour: (1) Set Inheritance mapping as JOINED, which means each subclass will have its own table. A table for a subclass contains only columns for the attributes, which are specific for this subclass. The inherited attributes of the superclass are stored in the table of the super-class. The primary key of the table of the super-class is also the primary key and foreign key of the table of the subclass, which forms an association of both tables. To retrieve a child object from the database the super-class and subclass tables are joined. Advantages of this approach are that there is only very few redundant data stored in the table and the created database schema is clear and structured, since it is almost 1-to-1 mapped from the object classes. (2) Set
FETCH_CONTAINMENT_EAGERLY (default: false) as true, means that containment relations are completely read in memory, because we want to load complete process at once.

We use the database component developed as a part of a BPEL fragment-mining prototype for another diploma thesis. For a more detailed introduction of the database component, reader is referred to [Ger08]. The database component works with MySQL or PostgreSQL database currently. If required, it can easily be extended to work with other database types due to the abstraction of Hibernate.

4.2.2 Graphbuild Component

The graph-building component is responsible for building rooted directed acyclic graph from BPEL-process. The class DefaultDirectedGraph\( \langle V, E \rangle \) of JGraphT is used as graph data structure to hold the generated rooted DAG pattern process graph. With Java Generics is possible to specify arbitrary type for \( V \). A class ActivityNode is created as datatype of \( V \). For edge \( E \), the standard implementation of JGraphT is used. ActivityNode has the following attributes: activity name, activity type, activity ID (which is the primary key of the activity in the database). ActivityNode has a set of subclasses. One subclass is called StructuredActivityNode, which represents structured activity. It has an additional attribute to determine whether a StructuredActivityNode object is a start node. For activity types, which have additional attributes to be searched, corresponding subclasses of ActivityNode are created. For example, class InvokeActivityNode has additional attributes such as Porttype, Operations.

Class ProcessFlowGraph represents a BPEL process graph. It contains following attributes: (1) process ID, process name, process namespace are used to hold information of BPEL-process stored in the database. (2) Process graph with the type DefaultDirectedGraph\( \langle ActivityNode, E \rangle \) and start node correspond to a rooted DAG, which is transformed from a BPEL-process. The graph type is stored in attribute processGraphType. (3) Two maps are used to store region encoding (see section 3.2.1) and SSPI index (see section 3.3.1) for process nodes. Alternative, we can use a table like data structure to store transitive closure.

Class ProcessFlowGraphFactory is an abstract class, which is responsible for creating a ProcessFlowGraph Object from a loaded BPEL-process from the database. The generated rooted DAG is stored in attribute processGraph. The transformation works as follows: it iterate through the BPEL-process, for each visited activity, nodes and edges are generated according to the mapping rules described in section 3.1.1. The mapping rule depends on activity type. For basic activity or each structured activity type, there is a method for the graph mapping for that activity type. There are three help classes (FlowLinkHelper, FlowLinkHelperBasicActivity, FlowLinkHelperStructuredActivity) for the
transformation of Flow activity, since it is complexer than transformation of other activity types. Class `NewStructuredProcessFlowGraphFactory` extends class `ProcessFlowGraphFactory`, and implements two abstract methods to handle `pick` and `if` activity.

![Class diagram for the Graphbuild module](image)

Figure 24: Class diagram for the Graphbuild module

### 4.2.3 Query Evaluation Component

Query evaluation component deals with evaluation of business process query under exact or inexact matching semantic. There are three submodules of the query evaluation component: Matching, DataStructure, Util. We introduce structures and contents of these submodules in the following.

#### Submodule Matching

The main part of the query evaluation component is the submodule Matching, which implements the graph matching algorithms introduced in chapter 3.
Class `ProcessQueryEvaluation` is the entry point (or interface) of the query evaluation component. Its method `evaluateProcessQuery` implements the algorithm `ProcessQueryEvaluation` in Listing 1. It analyzes the query graph and dispatches algorithms according to the search type. Besides query, there are some optional attributes: (1) attribute `filterProcesses` is used to specify filter criteria of the BPEL-processes to be matched. If it is not specified, all BPEL-processes stored in the database are compared with the query. (2) Attribute `searchType` specifies whether exact search or inexact search is desired, default value is exact search. (3) Attribute `minMatchingSimilarity` is threshold for matching results to be displayed, default value is \( \frac{1}{|Q|} \), where \( |Q| \) is number of edges of the input query graph, that is at least one matching edge is required.
For exact search, algorithm EvalQueryExactMatch in Listing 2 is used that is implemented in class EvalQueryExactMatch. It goes through all BPEL processes to be matched, transforms each BPEL-process to a rooted DAG, and performs graph matching between the query graph to each process graph, in doing so it dispatches to an adequate twig join algorithm according to query graph type, process graph type.

Class TwigStackAlgorithm is an abstract class. It implements all sub-procedures and sub-function of TwigStack Algorithm in Listing 3. All subclasses of class TwigStackAlgorithm must implement procedure twigStackExactMatch and procedure twigStackInexactMatch.

Class TwigStackAlgorithmBasic extends the class TwigStackAlgorithm. It implements: (1) method twigStackExactMatch, which implements TwigStack algorithm in Listing 3 for computing all exact matchings of a twig pattern query in a process tree. (2) Method twigStackInexactMatch, which is modified TwigStack algorithm (see section 3.2.4) for computing all inexact matchings of a twig pattern query in a process tree by using path based metric.

Class TwigStackAlgorithmForDAGQuery2 extends the class TwigStackAlgorithm. It implements TwigStackD2 Algorithm in Listing 5, which computes answers to a twig pattern query against a rooted DAG process graph. Class DagStackDAlgorithm extends the class TwigStackAlgorithmForDAGQuery2. It implements DagStackD Algorithm presented in section 3.3.4, which computes answers to a rooted DAG pattern query against a rooted DAG process graph.

For inexact search, algorithm EvalQueryInexactMatch in Listing 7 is used that is implemented in class EvalQueryInexactMatch. It goes through all BPEL-processes to be matched, transforms each BPEL-process to a rooted DAG, and performs inexact matching between the query graph and each process graph. After all BPEL-processes are processed, matching results are stored in a list. They are sorted by the matching similarity.

Class Query accepts a query in form of BPEL file, creates a QueryGraph Object. It checks graph type of the query. If query graph contains cycle, error message is displayed. If it does not have a root node, it is transformed to a rooted DAG as described in section 2.3. Class QueryGraph represents a query graph. Besides the DAG pattern query graph as attribute, it has additional attributes such as start vertex, query graph type. Query graph related operations described in section 3.2.2 and section 3.4.2 are implemented in this class.

Submodule DataStructure
Submodule DataStructure contains classes as data structures for the graph matching algorithms in chapter 3.

Class `NodeRegionEncoding` is positional representation of a process graph node with a tuple of three fields: (start; end; level) (see section 3.2.1). Class `Predecessors2` is the SSPI index introduced in section 3.3.1. Class `NodeInStack` represents a data element in the partial solution stack (see section 3.2.2) of a query node. Each element in the stack consists of a pair: (positional representation of a node from query node q's stream, pointer to a node in the parent stack of q). Class `PoolItem3Ext` and its inner class `PoolItem3ExtChildReferences` represent a pool Item introduced in section 3.3.3. Class `SolutionStream` represents a solution stream described in section 3.4.2. Class `StreamItem` and its inner class `StreamItemChildReferences` represent a stream element described in section 3.4.2.

Submodule Util

Submodule util includes all utility classes for searching. JGraph provide classes for some commonly used graph algorithms, which includes graph traverse algorithms. We need embed additional functionalities in graph traverse algorithms such as calculation region encoding. With the implementation of those
algorithms in JGraphT is not easily to extend the desired features. We implement the graph traverse algorithms again in another way. It has following classes: AbstractGraphTraverse, DepthFirstTraverse, DepthFirstTraverseExtension, BreadthFirstTraverse, GraphAnalyse (It checks graph types: a tree or a dag).

Figure 27: Submodule Util

All graph-traverse related classes are extended from the abstract class AbstractGraphTraverse, and must implement the abstract method traverse. DepthFirstTraverseExtension extends DepthFirstTraverse with some additional methods and attributes to compute region encoding for process graphs. DepthFirstTraverseExtensionSSPI extends DepthFirstTraverse can be used to compute SSPI index (see section 3.3.1) for process graphs.

Class GraphType is an enumeration class that contains some graph types used by the query processing.
Class **GraphAnalyse** is used to check some properties of query graph or process graph. It provides method to check whether a direct graph is acyclic, connected, rooted etc.

### 4.2.4 Possible Extensions

There are several ways to extend the prototype.

First of all the GUI component is not implemented in this thesis due to the time limitation. A GUI component interacts with end user and query evaluation component. It allows user to input a BPEL-process query either in BPEL file format or as a graph analogous to how BPEL-processes are typically designed. Wildcard or some special characters are allowed to appear in attribute of activities in query graph to match similar words. User can specify search type. A list of matching results or an empty list is returned by the query evaluation component after submission a query. Each matching result contains following information: BPEL-process ID, location, BPEL-process graph, matching subgraphs in the BPEL-process graph or other information. The results are displayed on the GUI in appropriate form.

Currently we transform BPEL-processes to DAGs at runtime, and compute region encoding, SSPI index or transitive closure for BPEL-processes at runtime. In order to improve performance, these steps can be done in pre-processing time. The data can be stored in some self-defined tables. These tables should keep “insync” with existing process tables, by update of process tables are these tables also updated.

The prototype is developed based on a normal relational database. Later, it should work with a BPEL repository to support vision control or other features. It should also be integrated with BPEL-process design tools.
Chapter 5: Conclusion and future work

In this work, we presented a graph based query language that allows users to query structural information of business processes visually. Through analyzing the data model of business processes, we know that business process specification can be modelled as an attributed directed acyclic graph. We examine the general requirements on querying business processes, and present the data model for the query language. In general, a business process query can be viewed as a business process fragment with activities having value-based predicates. Queries are written essentially the same way as business process specifications. Just like business process, a business process fragment can be represented as a directed acyclic graph. Therefore, the problem of querying business processes can be reduced to the graph pattern matching problem. In some use cases, users may wish to retrieve results which exact match the query. In other scenario, users may want to retrieve not only exact matching results but also results that are similar to the query. We defined two matching semantics (exact matching and inexact matching) that correspond to two search types (exact search and inexact search). To rank the inexact matching results, a new similarity metric based on connected component is defined. In order to answer such queries efficiently, we extended and developed exact matching algorithms and inexact matching algorithms. The solution is developed for business processes in general, but we also applied it to BPEL, which is an OASIS standard to specify and execute business processes based on Web Services. Through analyzing the language constructs in BPEL specification, we define the mapping rules to map BPEL-process to rooted DAG. A prototype is developed for querying BPEL-processes stored in a database. The prototype should be extended with a GUI component that is not in the scope of this thesis.

Our graph based query language is used to query structure information about business process specifications, not about verification of possible runs of business processes. Therefore, we ignore some business process constructs, which are related to runtime semantics, like transition condition and data connectors. In BPEL, we ignore variables, transition conditions and focus on the control flows, which define execution order of activities in business process. If these informations are desired to be searched, we can also extend the data model and algorithms. For transition conditions, we can extend the graph model for queries and business processes with labelled edges. For variables, we can map variable definitions to subgraph of the business process graph. Data connector is defined implicit in BPEL. We can create subclasses of ActivityNode to enable to search variable information stored in attributes of certain activity types (for example, assign activity).

It is also possible to use the graph based query language in combination with other query languages or methods in the business process area. We identified two possibilities: (1) as mentioned in section 2.3, our
query language is used to query business process specifications, not about possible runs of business processes. Querying the possible runs is of very high complexity. If the run verification is desired, our graph based query language can be used to “filter” business processes for the expensive verification processing. Because querying of specifications in fact “approximates” the querying of runs (e.g. only specifications that contain two given activities may potentially have runs where both occur). (2) We can use a query written in other query language to narrow the search space (for example, query the relevant metadata of business processes), then send a query in our query language to find more specific business processes. Because a BPEL-process is a Web Service (see section 2.1.2), it is possible to use frameworks or standards for discovering of Web Service to discover BPEL-processes. There are two possibilities to discover Web Services via meta-data: via a centralized registry (UDDI) or by distributed discovery (Metadata Exchange). For example, users may send a query with some metadata informations to find services of certain business categorie (for example “flight booking service”), after that run the graph based query to find more specific business processes that meet user’s need.
Bibliography


Harvey, Michael: *Essential Business Process Modeling*. Beijing; Köln [u.a.]: O’Reilly 2005


List of Figures

Figure 1: A sample business process.............................................................. 6
Figure 2: Example business process and Twig/Dag pattern queries ....................... 13
Figure 3: A Query without root node and its modified versions.............................. 14
Figure 4: A BP-QL query: find provided operations................................................ 19
Figure 5: (a) Negation (b) Path constraints. ...................................................... 19
Figure 6: Example graph, query, and exact matching results................................. 22
Figure 7: Example query .................................................................................. 23
Figure 8: A sample process graph, a query graph, and their common subgraphs ........ 28
Figure 9: An inexact matching example.......................................................... 30
Figure 10: Mapping a sample BPEL process onto a tree ....................................... 34
Figure 11: Tree representation of a sample XML document.................................. 38
Figure 12: Initial state of cursors and stacks of a sample query............................. 39
Figure 13: Compact encoding of answers using stacks ........................................ 40
Figure 14: Illustration of rooted DAG representation ......................................... 45
Figure 15: A query graph for the test ............................................................... 53
Figure 16: Solution pools for sample query graph and process graph ................... 55
Figure 17: An example for subgraph matching under AND- and Or-Matching Semantics ... 62
Figure 18: Initialized solution streams for example query graph and process graph .... 66
Figure 19: Compute maximal assignments for a subtree rooted at a non-leaf node in query tree .... 70
Figure 20: Compute maximal assignments for a subgraph rooted at a non-leaf node in a query DAG ...... 71
Figure 21: Processing of combination of subtree assignments............................... 78
Figure 22: System Architecture......................................................................... 85
Figure 23: Store a BPEL process in a relational database .................................... 86
Figure 24: Class diagram for the Graphbuild module........................................... 88
Figure 25: Submodule Matching......................................................................... 89
Figure 26: Submodule Datastructure................................................................... 91
Figure 27: Submodule Util............................................................................... 92
Index of Listings

Listing 1: Algorithm ProcessQueryEvaluation ................................................................. 36
Listing 2: Algorithm EvalQueryExactMatch ................................................................. 37
Listing 3: TwigStack Algorithm .................................................................................... 42
Listing 4: TwigStackD Algorithm .................................................................................. 50
Listing 5: TwigStackD2 Algorithm ................................................................................. 59
Listing 6: Procedure initSolutionStreams ................................................................. 66
Listing 7: Algorithm EvalQueryInexactMatch ............................................................ 69
Listing 8: Procedure computeMaxAssignForTreeNodes ........................................... 72
Listing 9: Procedure computeMaxAssignForDAGNodes ........................................... 76
Listing 10: Function maxMatchsOfQueryTree ............................................................ 79
Listing 11: Function maxMatchsOfQueryDAG ............................................................ 81
# Index of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP</td>
<td>Business Process</td>
</tr>
<tr>
<td>BPEL4WS</td>
<td>Business Process Execution Language for Web Services</td>
</tr>
<tr>
<td>BPEL</td>
<td>see BPEL4WS</td>
</tr>
<tr>
<td>BPM</td>
<td>Business Process Modelling</td>
</tr>
<tr>
<td>BPMI</td>
<td>Business Process Management Initiative</td>
</tr>
<tr>
<td>BPML</td>
<td>Business Process Modelling Language</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modelling Notation</td>
</tr>
<tr>
<td>BPQL</td>
<td>Business Processes Query Language (BPMI Standard)</td>
</tr>
<tr>
<td>BP-QL</td>
<td>Business Processes Query Language (A novel query language developed by Tel-Aviv University)</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>HQL</td>
<td>Hibernate Query Language</td>
</tr>
<tr>
<td>OASIS</td>
<td>Organization for the Advancement of Structured Information Standards</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery, and Integration</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XPath</td>
<td>XML Path Language</td>
</tr>
<tr>
<td>XPDL</td>
<td>XML Process Definition Language</td>
</tr>
</tbody>
</table>
Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, den 02. September 2008