Inf Syst E-Bus Manage (2015) 13:69–106
DOI 10.1007/s10257-014-0243-6

ORIGINAL ARTICLE

Graph theory and model collection management: conceptual framework and runtime analysis of selected graph algorithms

Dominic Breuker · Patrick Delfmann · Hanns-Alexander Dietrich · Matthias Steinhorst

Received: 15 May 2013 / Revised: 6 February 2014 / Accepted: 18 February 2014 /
Published online: 27 February 2014

Abstract Analysing conceptual models is a frequent task of business process management (BPM), for instance to support comparison or integration of business processes, to check business processes for compliance or weaknesses, or to tailor conceptual models for different audiences. As recently, many companies have started to maintain large model collections and analysing such collections manually may be laborious, practitioners have articulated a demand for automatic model analysis support. Hence, BPM scholars have proposed a plethora of different model analysis techniques. As virtually any conceptual model can be interpreted as a mathematical graph and model analysis techniques often include some kind of graph problem, in this paper, we introduce a graph algorithm based model analysis framework that can be accessed by specialized model analysis techniques. To prove that basic graph algorithms are feasible to support such a framework, we conduct a performance analysis of selected graph algorithms.

Keywords Business process modeling · Model collection management · Model analysis · Graph algorithms

D. Breuker · P. Delfmann · H.-A. Dietrich · M. Steinhorst (✉)
WWU Muenster - ERCIS, Leonardo-Campus 3, 48149 Muenster, Germany
e-mail: matthias.steinhorst@ercis.uni-muenster.de; Steinhorst@ercis.uni-muenster.de

D. Breuker
e-mail: Breuker@ercis.uni-muenster.de

P. Delfmann
e-mail: Delfmann@ercis.uni-muenster.de

H.-A. Dietrich
e-mail: Dietrich@ercis.uni-muenster.de

1 Introduction

As part of their business process management (BPM) activities, many organizations have created large collections of conceptual models (Rosemann 2006). These collections consist primarily of business process models. They may, however, also contain data models, organizational charts, or other kinds of models (Scheer 1992). Given the complexity of large organizations' businesses, it is not surprising that they use conceptual models in large quantities. Consider the process collection of Suncorp, which contains more than 6,000 models (La Rosa et al. 2013). Other examples include the BIT process library (~750 models) (Fahland et al. 2009) or the process collection of Dutch municipalities (~500 models) (Dijkman et al. 2011a). Recent studies indicate that process model collections may contain hundreds or thousands of models, each of which may in turn consist of hundreds or even thousands of elements (Rosemann 2006; Dijkman et al. 2012).

Against the backdrop of these numbers, practitioners have articulated a demand for (semi-)automatic model analysis support (Fauvet et al. 2010). BPM scholars have noticed this demand and proposed a plethora of different analysis techniques (Van der Aalst 2013). Dedicated model repositories, most notably AProMoRe (La Rosa et al. 2011a), aim at providing all the tools needed to manage and analyze large model collections in one place (Dijkman et al. 2012).

A well-known example of process model analysis for quality control is soundness checking. It is used to verify that a process is free of deadlocks and activities that can never be executed as well as that it always terminates properly (Van der Aalst 1998). Managing process models may, for example, involve activities such as merging business process models describing similar processes (e.g., as part of a post-merger integration project). In practice, this can be a very laborious and time-consuming task. For instance, merging a subset of the process model variants reported in La Rosa et al. (2013) required 130 man-h to get 25 % of the task done. The lion's share of time was spent on identifying similar regions in these models (La Rosa et al. 2013). Hence, algorithms searching for similar parts and merging them have been developed (La Rosa et al. 2010; Li et al. 2009).

While these and countless other BPM techniques indeed solve the problems they have been built to tackle, scholars have recently raised concerns that they might nevertheless not be perfectly in line with practitioners' needs (Houy et al. 2011). Many scholars perceive the provision of isolated implementations of BPM techniques as an impediment to adoption. For instance, Van der Aalst (2013, p. 30) states that "Currently, many prototypes are developed from scratch and "fade onto oblivion" when the corresponding research project ends. Moreover, it is often impossible to compare different approaches in a fair manner as experiments are incomparable or cannot be reproduced. Given the importance of BPM, these weaknesses need to be tackled urgently."

Motivated by these critical voices, we propose to add an additional layer of abstraction to future BPM management and analysis software. Agreeing with Houy et al. (2011), we observe that most of the BPM techniques put forth by scholars concentrate on a particular scenario, such as merging two EPCs. Agreeing with Van der Aalst (2013, p. 30), we also observe that, if a corresponding implementation is

provided, it likely is an isolated tool serving exactly this purpose. The effect is that a potential user is presented with a black box piece of software that s/he may either use as it is or not at all. However, we cannot expect to anticipate all the users' needs. Consequently, we must provide users not with black box procedures but flexible techniques which they can adapt themselves to the problems they face in their particular situations.

Say, for instance, a practitioner got his hands on a soundness checker using Petri net reduction rules. Empirical studies demonstrate that the number of different modeling languages used in practice is huge and, even worse, that many organizations modify languages or design their own ones from scratch (Becker et al. 2010a). Consequently, the practitioner's organization is likely not using Petri nets. Even though the idea of checking soundness by reduction rules is transferable to other notations [e.g., EPC (Mendling et al. 2007) or YAWL (Wynn et al. 2009)], doing this in practical settings will usually be prohibitively expensive because the practitioner needs to develop a deep understanding of the source code. Hence, it takes a researcher publishing a new tool to deal with a new notation.

To enable a user to open the black box himself, we have to make sure that what he finds inside does not overwhelm him. We thus propose to put a new layer of abstraction in between the source code and the BPM technique. On the one hand, the level of abstraction must be low enough to provide the flexibility needed to implement state-of-the-art BPM techniques using this layer. On the other hand, the level of abstraction must be high enough to make the implementation easy to understand and modify.

Our proposal is that concepts from graph theory could be versatile tools in providing this layer of abstraction. The central object in graph theory is the mathematical notion of a graph, consisting of nodes and edges whose semantics can be described by labels. Effectively, any conceptual model can be described as a graph, with the meanings of all modeling elements (type, description, etc.) being encapsulated within the labels. Graph theory provides numerous abstract problems on graphs. An example is subgraph isomorphism (SGI), which deals with finding structural patterns. Using a layer of graph algorithms as an intermediate abstraction, users trying to understand and adapt a BPM tool need to worry less about algorithmic subtleties of model processing and could focus more intensively on the conceptual ideas behind the tool.

However, many graph problems of interest are computationally intractable (Garey and Johnson 1979). Despite this fact, considerable research effort has been put into developing algorithms with acceptable runtime performance in practical usage scenarios (Cordella et al. 2004; Ullmann 1976). With model repositories growing rapidly (Dijkman et al. 2012), we argue that runtime performance is of paramount importance to guarantee applicability in practice. Runtime evaluations of graph algorithms have been conducted in disciplines other than BPM (e.g., in computational biology Marc et al. 2005; Nijssen and Kok 2006; Welling 2011). As the results cannot be transferred automatically to conceptual models relevant in the context of BPM, we conduct extensive runtime evaluations to fill this gap.

The purpose of this paper is threefold. First, we propose an architecture for a model management and analysis framework of which graph algorithms are a central

component. End users develop BPM approaches using a flow-based programming language in which abstract functionality—encapsulated in *bricks*—is put together in a modular construction system. Second, after identifying graph algorithms as an important class of such functionality, we extract four graph problems that frequently occur as part of model management and analysis techniques from BPM literature reviews. These graph problems address structural (SGI, frequent pattern detection, and maximum common subgraph detection) and behavioral (state graph minimization) aspects of models. Third, we empirically assess the runtime performance of algorithms for these problems on large collections of conceptual (process) models.

The paper at hand is an extension of our earlier work on this topic (Becker et al. 2012b). It extends the findings as follows:

- We conceptually specify a graph algorithm-based model analysis framework.
- We extend our runtime evaluation to include algorithms for the problems of maximum common subgraph detection and state graph minimization.
- We extend our analysis of algorithms for frequent pattern detection to include a detailed discussion of the memory requirements of these algorithms.

The remainder of the paper is structured as follows. In Sect. 2, we introduce the architecture of a graph algorithm-based model management and analysis framework. To illustrate the use of the framework, we provide an exemplary use case scenario. To identify typical graph problems that occur during model management and analysis, we conduct a literature survey in Sect. 3.1. We discuss specific BPM approaches in more detail in order to explain how they could incorporate algorithms for these problems (Sect. 3.2). In Sect. 4, we conduct an extensive runtime analysis of corresponding graph algorithms on collections of conceptual models. We discuss limitations of our work in Sect. 5. The paper closes with a brief summary and an outlook to future research in Sect. 6.

2 A graph algorithm-based model management and analysis framework

We envision a framework that allows users to develop a high-level management and analysis approach by modeling the main analysis steps using a flow-based programming language. Such flow-based programming languages have, for instance, been used in the area of data integration for modeling and enacting ETL processes (extraction, transformation, loading).[1] The user shall be able to model the process of model analysis using a set of predefined *bricks*. These bricks provide abstract, reusable functionality and can be connected to other bricks by means of standardized data structures. A brick works on its input data to provide output data. Bricks may have any number of parameters to be set by a user.

The rationale for developing model analysis and management approaches this way is twofold. One advantage is that a user's understanding of an approach can be facilitated as s/he is provided with a visual representation of its logic. Far more important though is that the approach is easily customizable. Doing modifications

[1] See http://de.talend.com/download for an example of such software.

boils down to either using different bricks or modifying a brick's options. If standardized data structures such as labeled graphs are used, a BPM approach could, for instance, be used with any kind of modeling language by customizing the approach using a graphical environment with high-level functionality.

To exemplify how such a framework could be used and what the level of abstraction we envision is, consider the example illustrated in Fig. 1. Say a user wants to implement an analysis approach to check process models for violations of regulatory compliance rules. Such violations can be found by searching for patterns representing violations (Bräuer et al. 2013). The first step in the implementation is to load both models and patterns. Corresponding bricks would be used to provide them as lists of graph objects (Brick 1 and Brick 2). The nodes and edges of these graphs carry particular attributes capturing the domain-specific semantics of the model element (type, label, etc.). The lists of graphs loaded by the bricks can be used as input for other bricks now. In this example, the analyst is interested only in processes that are from the risk management area and not too small. To accomplish this, Brick 3 is used to filter the list such that only graphs with at least 10 nodes are retained that also have an attribute *Process Area* whose value is *Risk Mgmt*. This is configured in the graphical user interface using the brick's options window.

The heart of the approach—pattern matching—is then implemented by adding Brick 4. Pattern matching can, for instance, be accomplished by SGI algorithms (Ullmann 1976) (see more details in Sect. 3.1). By calling an SGI algorithm from an

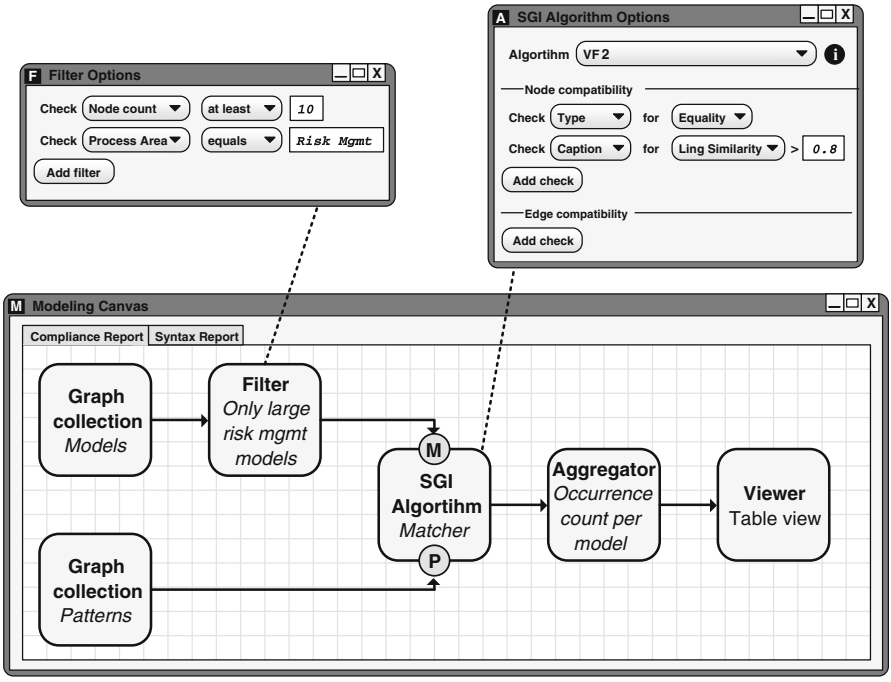

Fig. 1 A compliance checker implemented in the flow-based programming canvas

underlying library of graph algorithms, Brick 4 searches, for every combination of model and pattern graphs, for all occurrences of the pattern. It outputs a list of tuples, each consisting of the model graph, the pattern graph, and a list of all the occurrences. Again, the user can customize the brick using options. In a next step, the pattern search results may need to be transformed for presentation. To that end, an aggregator brick (Brick 5) may be used. In this example, the brick takes the tuples returned by the SGI algorithm brick as input, groups them by the model graphs, and counts the pattern occurrences for each model (over all patterns and their occurrences). The results are fed to a viewer brick (Brick 6), which displays the results in a table.

It is important to note that the framework we describe serves illustrational purposes only. Neither do we want to propose a full list of components the framework should have nor do we know all the bricks it would need. Indeed, we believe that only through extensive empirical work, most importantly case studies, such a list could ever be devised. Nevertheless, we identified certain areas deserving attention and perceive the area of graph algorithms to be in most pressing need. This is because graph problems are often computationally intractable. Hence, there is no guarantee corresponding algorithms will perform well when applied to conceptual models. To establish empirical results on their performance is therefore this study's goal. Ultimately, the results shall be used to guide the selection of algorithms for the framework.

3 Model management, model analysis tasks, and related graph problems

3.1 Overview of literature

We now identify relevant graph algorithms that occur frequently in model management and analysis approaches from the BPM literature. They are meant to be examples of algorithms that could populate the *graph algorithms* component of the framework discussed in Sect. 2. We start with several literature reviews in the field of BPM, synthesize basic model management and analysis tasks, discuss corresponding BPM approaches, and finally show how they relate to one or more graph problems. Where appropriate, we extend our discussion of BPM approaches (that naturally have a strong focus on process models) to include model analysis techniques put forth in the domain of database management. We do not want to claim that all BPM approaches involve graph problems, but only that a significant number of them does. We also do not want to claim that the graph problems we discuss are all one may ever need in BPM approaches. There likely are many others, but as we demonstrate, widespread support of BPM model analysis and management activities can already be achieved with these. The purpose of this section is to demonstrate the applicability of graph algorithms in the domain of conceptual model management. In some cases, there may be other algorithmic approaches that can be used to address specific model management and analysis tasks as well. For example, translating models may be supported by graph parsing algorithms or SGI-based approaches (see details below).

The literature reviews we draw upon are a survey by Van der Aalst (2013) who analyzed 289 papers and defines a set of twenty BPM use cases involving conceptual models, a survey by Dijkman et al. (2012) who define nine activities for managing large collections of business process models, and a survey by Wang et al. (2013) who focus on model query approaches. These surveys and the literature cited therein constitute the basis of our analysis. We used them to synthesize a set of seven basic model management and analysis tasks. We ignored all tasks that need to be performed before as well as after managing and analyzing models. For example, tasks related to model creation (either modeling or event-log analysis) have been discarded as they take place before model analysis. Tasks such as those related to model execution (e.g., monitoring execution in workflow management systems) take place after model analysis and have been discarded too. After synthesizing the remaining tasks, we ended up with *querying models*, *finding similar models*, *merging models*, *managing model variants*, *translating models*, *refactoring models*, and *verifying models*. In addition to literature on BPM, which focuses mostly on process models, we also considered sources related to data models. These have been selected from our own experience.

Querying models refers to searching for occurrences of a particular model (fragment) within a given collection. A number of model query languages have been proposed recently. Most notably, BPMN-Q is an SQL-based query language to detect fragments in BPMN models (Awad 2007; Awad and Sakr 2010; Awad et al. 2008; Sakr and Awad 2010). As demonstrated by Awad and Sakr (2010), using SQL to query graph structures can lead to serious performance problems because node and edge tables may need to be joined very often. Indexing can be used to narrow down the search set and increase performance (cf. BeehiveZ Jin et al. 2010, 2011a, b, 2012). Other model query languages include BP-QL (Beeri et al. 2005, 2006, 2008; Deutch and Milo 2007) for the business process execution language (BPEL), WISE (Shao et al. 2009), VisTrails (Scheidegger et al. 2008), AFSA (Mahleko and Wombacher 2006), LM + SM (Qiao et al. 2011), BPQL (Momotko and Subieta 2004), and OPSIM (Lincoln and Gal 2011). Querying conceptual models is also relevant in the domain of database management. Data model patterns that can be searched for in collections of data models are presented in the works of Fettke and Loos (2005) and Batra (2005). From a graph theoretical point, the task of querying models to detect particular fragments relates to the *SGI* problem. Algorithms for SGI are concerned with detecting exact occurrences of a pattern graph (the model fragment) within a model graph (the model to be searched in). SGI algorithms find structure-preserving mappings between the nodes of the pattern and subgraphs of the model graph. Structure is preserved if adjacent pattern nodes are mapped to adjacent counterparts in the model graph.

Finding similar models mainly consists of two steps. First, a notion of similarity must be defined. Three types of similarity metrics can be distinguished (Dijkman et al. 2011a). Node matching similarity takes the labels of model elements into account. Structural similarity also considers a model graph's structure. Behavioral similarity takes execution semantics into account. Approaches for data models exist as well (Fettke and Loos 2005). Further work along these lines has been proposed in Dumas et al. (2009), Kunze et al. (2011), and Kunze and Weske (2011). The second

step is defining efficient mechanisms to retrieve similar models. Small, characteristic patterns to estimate the similarity of two process models are defined in Yan et al. (2012). From a graph theoretical point of view, the problem of determining the structural similarity of model graphs could be supported with algorithms for *maximum common subgraph detection*. These algorithms take a set of graphs as input and determine the largest subgraph that is contained in all input graphs (Koch 2001). In other words, they find a common subgraph such that no other common subgraph has a greater number of distinct edges. The idea is to measure similarity by the size of the largest common subgraph. Another idea is to use *SGI* algorithms for finding patterns as proposed in Yan et al. (2012). State graphs are a means of determining the behavioral similarity of two models (Becker et al. 2012a). As working with state graphs can be computationally expensive due to the state space explosion problem, algorithms finding a *minimal state graph* speed up the matching process. Algorithms for this problem construct a state graph that accurately captures the behavior of a process but is minimal with respect to the number of nodes of the state graph (Hopcroft et al. 2008).

Merging models refers to combining a set of input models to one integrated model, for example, to consolidate processes after mergers and acquisitions. A number of different approaches have been proposed that subsume the behavior of all input models (La Rosa et al. 2010, 2013; Gottschalk et al. 2008a; Reijers et al. 2009), consider textual similarity in the merging algorithm (Sun et al. 2006; Mendling and Simon 2006; Li et al. 2010), or present merging mechanisms for particular modeling languages (Gottschalk et al. 2008a; Reijers et al. 2009; Sun et al. 2006; Mendling and Simon 2006; Li et al. 2010). Merging models is also an important task in database management, where it is referred to as schema matching. A comprehensive survey of corresponding approaches can be found in Rahm and Bernstein (2001). From a graph theoretical point of view, algorithms for *maximum common subgraph detection* can be used as components in a model merging mechanism. For instance, the merging algorithm proposed by La Rosa et al. (2013) contains a sub-procedure to detect maximum common regions in process models (see more details in Sect. 3.2).

Managing model variants is necessary if different stakeholders have different information needs. Managers may be interested in a high-level overview of the business processes, whereas an IT specialist may require more detailed process descriptions. To account for these needs, a variant for each stakeholder can be created. Two strands of research can be distinguished. In the first one, a consolidated model is maintained from which variants are derived according to stakeholder preferences. A technique to configure process models by means of parameters is presented in Rosemann and van der Aalst (2007). The approach includes a mechanism to check the resulting variant for inconsistencies like deadlocks or livelocks. Other work falling into this category is presented in Gottschalk et al. (2008b), La Rosa et al. (2011b), Hallerbach et al. (2009). In the second strand of research, process variants are not consolidated but stored separately. Corresponding approaches provide mechanisms to retrieve variants and keep track of changes. A pattern-based querying approach that allows for finding process model variants that are similar to a given query is proposed in Lu et al.

(2009). Other work that falls into this category is presented in Pascalau et al. (2011), Derguech et al. (2010), Ekanayake et al. (2011), Weidlich et al. (2011). From a graph theoretical point of view, algorithms for *maximum common subgraph detection* can be used to find similar parts of process variants that need to be consolidated. As deadlocks can translate to particular patterns within the model graph, algorithms for *SGI* can be applied to check a variant derived from a configurable model for such inconsistencies. An example of a deadlock pattern requiring an isomorphism check is a model fragment consisting of an XOR split with two outgoing branches each holding an activity and a subsequent AND join. Querying approaches as proposed in Lu et al. (2009) can also be supported by algorithms for SGI. If model behavior needs to be processed, algorithms for *state graph minimization* speed up the model checking procedures.

Translating models refers to transforming a model developed with one modeling language into a model of a different one, for example, to translate BPMN models to BPEL (Ouyang et al. 2008). In Ouyang et al. (2008), the authors identify characteristic BPMN patterns that correspond to chunks of BPEL code. Examples are the repeat pattern (a loop in the control flow) or the sequence pattern (a sequence of activities). Similar work is presented in García-Bañuelos (2008). From a graph theoretical perspective, this model management task can be supported by algorithms for *SGI* used to identify the patterns to translate.

Refactoring models refers to detecting and resolving unnecessary complexities in model collections. This includes detecting clones in models (Dumas et al. 2013). A clone represents a particular model fragment appearing frequently in a model collection. Its presence may indicate undesired redundancy. Approaches to detect these clones are presented in Dumas et al. (2013), Ekanayake et al. (2012). An approach to reduce the complexity of EPC models is proposed in Polyvyanyy et al. (2010). It defines particular patterns called abstractions that are characteristic for EPC models. The idea of this work is to identify such abstractions and replace them with one EPC function, thereby reducing the overall complexity of the input model. Similar work is presented in Reijers et al. (2011), Weber et al. (2011), Dijkman et al. (2011b). From a graph theoretical point of view, the work discussed here is concerned with *identifying frequently occurring patterns* in model graphs.

Verifying models refers to determining syntactical or behavioral properties of models. As far as process models are concerned, this includes checking them for syntactical correctness (Mendling et al. 2008) or soundness (Van der Aalst 1998; Aalst et al. 2010). Model checking techniques can be applied to check these properties. They require a process model to be transformed into a state graph first (Feja et al. 2011). Especially for concurrent processes, state graphs can grow very large. Thus, minimizing their size is important. From a graph theoretical point of view, this can be supported by algorithms for *state graph minimization*. Verifying models may also include searching for model fragments that point at improvement potential or violations of compliance rules. Pattern matching plays an important role to accomplish these tasks. Typical weakness patterns that occur in business processes of the financial industry are defined in Becker et al. (2010b). Compliance checking is concerned with detecting violations to laws and regulations in process

models by means of model fragment search (Knuplesch et al. 2010; Awad et al. 2008). Again, algorithms for *SGI* can support these model management tasks.

This selection of literature demonstrates that there are a great number of frequent model management tasks that involve some form of graph theoretical problem. Using graph algorithms as a component in higher-level model management approaches yields great potential because they are modeling language-independent and can be transferred to many different model management tasks. Table 1 contains a summary of the literature analysis. It highlights which papers belong to a given model management task and which graph algorithms can be used as a component within a BPM approach addressing this task. The table demonstrates that there is a set of reoccurring graph theoretical problems contained in many different model management tasks. This section represents only a broad overview of these tasks and corresponding graph theoretical problems. We will provide a more detailed description of how graph algorithms can be used to manage large model collections in the following subsection. For each of the four graph problems we identified in this section, we will discuss at least one exemplary BPM approach from the literature and explain in more detail why the former can be interpreted as a part of the latter.

3.2 Particular examples

La Rosa et al. (2013) propose an approach to merging business process models that could incorporate a graph algorithm for maximum common subgraph detection. The approach takes two or more process models as input. We explain the approach for the special case of two input models. It works as follows:

- The first step processes all elements of the models that carry textual descriptions (such as events and functions in EPCs). The goal is to determine how similar pairs of such elements are. The first element of a pair is taken from one input model and the second from the other. Pairs are processed only if their types are equal (i.e., EPC functions would not be paired with EPC events). For each such pair, a similarity measure is applied that processes the textual descriptions.
- The second step is about establishing the similarity of all elements not carrying textual descriptions (such as connectors in EPCs). Again, pairs of such elements are only processed if their types are equal (i.e., EPC split connectors would not be paired with join connectors). As there is no textual description, the similarity of each pair is calculated as a function of their predecessor and successor events and functions (directly and transitively, i.e., across further connectors).
- In the third step, a mapping between elements from one model and elements from the other is created. From all possible mappings between such nodes, the one with the highest score is chosen. Hence, this step is about solving a combinatorial optimization problem for which the similarities calculated in the previous steps serve as inputs. It is set up such that elements in one model may end up with no element in the other to which it is mapped.
- In the fourth step, so-called "maximum common regions" are computed which are cohesive mapped subsections of the input models. In order not to construct confusing control flows by accident, elements in maximum common regions are

Table 1 Model management tasks, their related papers, and related graph problems

Model management and analysis task	Paper	Related graph problem
Querying models	Awad (2007), Awad and Sakr (2010), Awad et al. (2008), Sakr and Awad (2010), Jin et al. (2010), Jin et al. (2011a, b, 2012), Beeri et al. (2005, 2006, 2008), Deutch and Milo (2007), Shao et al. (2009), Scheidegger et al. (2008), Mahleko and Wombacher (2006), Qiao et al. (2011), Momotko and Subieta (2004), Lincoln and Gal (2011)	Subgraph isomorphism
Finding similar models	Dijkman et al. (2011a), Fettke and Loos (2005), Dumas et al. (2009), Kunze et al. (2011), Kunze and Weske (2011), Yan et al. (2012), Becker et al. (2012a)	Maximum common subgraph detection, state graph minimization, subgraph isomorphism
Merging models	La Rosa et al. (2010, 2013), Gottschalk et al. (2008a), Reijers et al. (2009), Sun et al. (2006), Mendling and Simon (2006), Li et al. (2010)	Maximum common subgraph detection
Managing model variants	Rosemann and van der Aalst (2007), Gottschalk et al. (2008b), La Rosa et al. (2011b), Hallerbach et al. (2009), Lu et al. (2009), Pascalau et al. (2011), Derguech et al. (2010), Ekanayake et al. (2011), Weidlich et al. (2011)	Maximum common subgraph detection, state graph minimization, subgraph isomorphism
Translating models	Ouyang et al. (2008), García-Bañuelos (2008)	Subgraph isomorphism
Refactoring models	Dumas et al. (2013), Ekanayake et al. (2012), Polyvyanyy et al. (2010), Reijers et al. (2011), Weber et al. (2011), Dijkman et al. (2011b)	Frequent Pattern Detection
Verifying models	Van der Aalst (1998), Mendling et al. (2008), Aalst et al. (2010), Feja et al. (2011), Becker et al. (2010), Knuplesch et al. (2010), Awad et al. (2008)	State graph minimization, subgraph isomorphism

removed from the mapping if they are situated at the beginning or at the end of the control flow in only one input model but not in the other. Mapping such nodes could lead to inappropriate control flows in the merged model.

- In the fifth step, a configurable merged process model, the output model, is built based on the mapping. For each pair of elements in the mapping, an element is added to the output model. All unmapped elements from either of the input models also become elements in the output model. If an edge connects two mapped elements in one input model and their counterparts in the other one are connected, one edge between corresponding elements in the output model is added. All other edges of either input model are added to the output model individually.

The first three steps of this BPM approach are about determining a mapping between elements from two models. This mapping is created using similarity measures and an optimization algorithm. Hence, they cannot be supported by graph algorithms. In our model management and analysis framework, these steps would have to be implemented in form of one or more bricks which we did not describe yet. In step four though, maximum common regions are constructed using a tailor-made algorithm. Instead, a *maximum common subgraph* algorithm could be used in a simple, iterative scheme. In each step, one searches for the largest common subgraph and removes all its nodes from the mapping temporarily to make sure it is not found in subsequent steps. The procedure terminates once no common subgraphs are found anymore, and the mapping is restored then. This way, all maximum common regions are found, which allows for proceeding with step five.

Dumas et al. (2013) introduce an approach to clone detection in business process model repositories. The idea of this approach is to identify and subsequently reuse identical process model sections. Having identified these frequently occurring process model sections—called *exact clones*—every new process model also containing such clones could reuse the already existing ones. The goal is to increase the maintainability of the process model repository. This clone detection approach could be supported by a *frequent pattern detection* algorithm. Exact clones in process model repositories can be understood as frequent patterns in graph collections. Hence, at least the first step of the clone detection approach—identifying clones in the process model collection—could be supported by a frequent pattern matching algorithm. Obviously, it must be taken care to map only pairs of compatible nodes and edges using appropriate checks for equivalence.

The approach put forth in Jin et al. (2010) is an example of a model query approach that directly uses the SGI algorithm by Ullmann (1976). The authors filter the set of models to be searched and use an adapted version of the Ullmann algorithm to detect patterns. The filter is built such that it keeps track of the labels appearing in the models. When processing a query, the list of models the Ullmann algorithm is executed on is narrowed down by removing models not having all the labels found in the query pattern.

State graph minimization is a pre-processing step for approaches to model checking which make use of state graphs. In particular, many approaches on business process compliance checking transform business process models into state graphs and check them for compliance by using, for instance, linear temporal logic (LTL) statements (like, e.g., an extension of BPMN-Q, cf. Awad et al. 2008). As state graphs generated from process models can reach a considerable complexity, LTL or related approaches often suffer from state space explosion. State graph minimization can help minimizing the complexity of queries.

4 Runtime experiments

4.1 Method and scope

Having identified a set of graph problems that frequently occur when managing and analyzing conceptual models, we now investigate how quickly particular graph algorithms for these problems return results on various collections of conceptual models.

The purpose of this study is twofold. On the one hand, we want to identify the extent to which the algorithms can be used efficiently within the framework we envision. On the other hand, we want to assess how well the algorithms' runtimes can be predicted using linear regression models. In doing so, we gain insights into the effect of particular inputs (e.g., model size) on the overall runtime performance. In addition to that, predictors could be integrated into the framework to provide users with runtime estimates.

We distinguish between structural and behavioral model management and analysis approaches. A structural approach can be supported using algorithms for SGI, frequent pattern detection, and maximum common subgraph detection. A behavioral approach can be supported using algorithms for state graph minimization. Algorithm selection for each problem was based on whether we could obtain publicly available implementations for the corresponding algorithms.

In terms of *SGI* we used the Ullmann (1976) and VF2 algorithms (Cordella et al. 2004). The Ullmann algorithm determines a set of mapping matrices describing all possible ways to map pattern nodes to model nodes. This set of matrices is then filtered to determine all isomorphic mappings. VF2 is based on the idea of iteratively augmenting a partial mapping solution under feasibility constraints. In terms of *frequent pattern detection*, we used the gSpan (Yan and Han 2002) and Gaston (Nijssen and Kok 2005) algorithms. The gSpan algorithm assigns a canonical label, called the dfs-code (depth first search), to each of the input graphs. This label is used to restrict the possibilities for extending a pattern. To reduce memory requirements, gSpan updates an appearance list containing only those graphs that contain a given pattern. In contrast, the Gaston algorithm stores the concrete mappings of pattern nodes to model nodes. This speeds up the isomorphism check but may lead to large memory requirements. In terms of *maximum common subgraph detection*, we used the Koch (2001) and McGregor (1982) algorithms. The algorithm by Koch exploits the work of Levi (1973). Koch proves that the maximum common subgraph problem can be transformed into the problem of finding the maximum clique in the vertex product graph (see Levi 1973 for more details) of the two input graphs. The McGregor approach is a backtracking algorithm that iterates through the two input graphs and maps nodes with compatible labels under structural feasibility constraints (cf. McGregor 1982 for more details). In terms of *state graph minimization*, we used the algorithms of Huffman (1954) and Hopcroft (1971). They remove unreachable states and iteratively merge equivalent ones. Furthermore, we analyze the algorithm of Brzozowski (1962), which reverses the state graph, determinises it, and then reverses again. While determinising a state graph may increase its size if it was nondeterministic before, determinisation is beneficial for many algorithmic tasks on state graphs, in particular model checking (Vardi 2007).

We applied these algorithms to several collections of models and measured their runtimes as well as memory requirements. For the SGI algorithms and the frequent pattern miners, we considered two different types of application scenarios. In one scenario, we considered only the types of model elements as labels (e.g., "function" in EPCs). In the other scenario, we additionally considered the elements' captions (e.g., "check invoice" for an EPC function). For elements with no caption, we always used their types as labels. As the algorithms use labels to reduce the search

space, we expected the algorithms to perform better in the scenario considering more detailed labels (type and caption).

With respect to the dynamic behavior of processes models, our goal is to prepare analyses of a process's state space by obtaining a minimum size state graph using one of the three algorithms. For this analysis, we considered runtimes for performing these minimization operations an important factor. State graphs are obtained from Petri nets whose formal semantics easily allows for generating these graphs. To achieve varying sizes, we generated state graphs under the assumption of varying k-boundedness. K-boundedness means that a place can hold at most k tokens. For instance, for $k = 1$, we generated a reachability graph containing exactly the part of the Petri net's state space that does not violate the 1-boundedness property. Experiments have been conducted on state graphs obtained under 1-, 2-, and 3-boundedness. In case of highly concurrent processes, memory usage can also be an issue for such kind of analysis. Out of memory exceptions may then occur during state graph creation from Petri nets, or at any time during execution of any of the minimization algorithms. Hence, we also measured the number of times the Java virtual machine (JVM) was claiming more than 80 % of available memory. We then aborted state graph creation or any minimization algorithm in these cases.

In order to predict the runtimes of the algorithms with a linear regression model, we first determined candidate regressors, inspired by the worst case complexity if known. In a next step, we inspected the data for outliers. If the distribution of the data was skewed due to the outliers, we performed a logarithmic transformation to alleviate this problem. If we did, we say so in the experiment's result section. Furthermore, we did a visual inspection of scatter plots to choose the structure of the regression model. By applying the resulting regression model, the coefficients of determination (R^2) for each of the regressions and their respective significance levels were calculated. The coefficient of determination equals the fraction of variance that was explained by the regression model. Hence, they range from 0 (no variance explained) to 1 (perfect relationship). We use a single * to indicate that the corresponding value is significant to the 0.10 level. ** Will indicate significance to the 0.05 level.

In order to see if the linear regression model is overfitting, we randomly split the data into a training set of 80 % and a test set of 20 % of the data. We then trained the estimator with the training set and computed the R^2_{Train}. Using the trained estimator, runtimes for the test set were estimated and compared to the true values, yielding R^2_{Test}. By comparing R^2_{Train} and R^2_{Test}, we check if the regression model is overfitting. If we do not see serious drops in variance explained when applying the estimator to the test data, then no overfitting happened.

4.2 Model base

We used ten model collections for our runtime experiments. The first one is the SAP reference model (Curran and Keller 1998) (SAP_EPC) containing 604 EPC diagrams. We obtained two additional model collections during modeling projects in industry. These collections (PA1_EPC and PA2_EPC) both describe the

processes of public administrations. They contain 2,200 and 604 EPCs. We also use two collections of organizational charts (PA1_ORG and PA2_ORG) from these public administrations, containing 88 and 18 models. Another collection (PA1_TTM) contains 491 technical term models (TTM) from the first of the public administrations. Becker and Schütte (2004) present a reference model for the retail industry. It contains 54 EPCs (RI_EPC) and 33 ER diagrams (RI_ERM), which we also use. Moreover, we use the common warehouse meta-model (Object Management Group 2003) containing 30 UML class diagrams (CWM_CD). Another model collection (MM_ERM) contains 35 ER diagrams from a meta-modeling project. Finally, we also use the BIT process library (Fahland et al. 2009), which contains three collections of Petri nets with 282 (BIT_A_PN), 421 (BIT_B3_PN), and 32 nets (BIT_C_PN).

Table 2 summarizes key characteristics of our model collections. The table contains information on the number of models in each collection as well as information on the average model size (Avg.), the standard deviation (Std.) and the minimum (Min.) and maximum (Max.) number of nodes and edges of a model in a given collection. The table demonstrates that the collections contain both very small models with an average size of 10 nodes and 10 edges and very large models with an average size of about 200 elements (nodes plus edges). The smallest model contains only two nodes and one edge, whereas the largest model we analyzed contains 912 nodes and 935 edges. We argue that most conceptual models exhibit sizes that fall into this range.

4.3 Patterns

To evaluate the Ullmann and VF2 algorithms for SGI, we needed not only models but also patterns to be searched for. We used the gSpan algorithm to generate these pattern collections with the minimally possible support subject to memory constraints. The support values range between 50 and 0.5 % depending on the model collection. In contrast to randomly generated patterns, using frequent patterns mined by gSpan ensures that we used patterns that actually exist in the model collections. This allows analyzing the runtime behavior of the algorithms when large amounts of patterns are indeed found, thus improving the validity of the study.

Key characteristics of our pattern collections are summarized in Table 3. For each of the model collections, two pattern collections were generated. The first collection contains patterns with nodes labeled using their element types only (*type* section of Table 3). The second collection contains patterns in which nodes are labeled with type and textual description (*caption* section of Table 3). For each collection, the table contains information on the Avg., Min., and Max. number of pattern nodes and edges, as well as on the Std. We also report the number of patterns obtained from each collection (# of patterns). For instance, the pattern collection corresponding to the SAP_EPC model collection labeled by types contains 399 patterns, with an average number of 4.30 nodes and 3.31 edges per pattern. The largest pattern of the *type* collections contains 41 elements as measured by the number of nodes and edges. The largest pattern of the *caption* collections contains 33 elements. These largest patterns are therefore roughly the same size as a small to

Table 2 Model base

Collection	# Of models	Nodes				Edges			
		Avg.	Std.	Min.	Max.	Avg.	Std.	Min.	Max.
SAP_EPC	604	20.74	18.74	3	130	20.80	20.84	2	138
PA1_EPC	2,200	100.95	109.33	2	912	102.19	112.95	1	935
PA1_TTM	491	9.76	12.55	2	156	8.48	11.73	1	155
PA1_ORG	88	9.39	14.00	2	98	7.33	9.83	1	52
PA2_EPC	604	45.69	34.70	4	227	47.35	38.05	2	246
PA2_ORG	18	13.56	18.53	3	63	13.22	19.12	2	62
RI_EPC	54	39.11	17.91	10	100	42.28	20.56	10	110
RI_ERM	33	20.55	8.02	8	34	46.67	17.96	16	78
CWM_CD	30	7.27	4.86	2	24	12.73	10.87	2	48
MM_ERM	35	35.00	16.36	15	79	72.17	34.56	28	164
BIT_A_PN	282	77.03	45.44	11	275	95.04	58.63	11	355
BIT_B3_PN	421	85.27	82.83	7	452	101.21	102.10	6	572
BIT_C_PN	32	129.59	128.62	17	546	135	135.43	18	572

medium sized model. We argue that this pattern size is ideal for runtime experiments because in typical applications of pattern matching in conceptual models the pattern is smaller than the model (cf. Sects. 2, 3). Furthermore, if a pattern is larger than the model it is searched in, both the VF2 and Ullmann algorithms will instantly abort as such a pattern cannot possibly be contained in the model. We conclude that the patterns we generated from the model collections are well suited for our runtime experiments.

4.4 Technical setup

To run the two algorithms for SGI on the model base, we used the publicly available VFLib graph matching library. It contains C implementations for both the Ullmann and the VF2 algorithm.[2] For gSpan and Gaston we used the Java library ParSeMis, which is also publicly available.[3] For the maximum common subgraph algorithms by McGregor and Koch we used the implementations as provided in Welling (2011). State graph minimization was performed using the BRICS automaton package,[4] which is a java package offering an efficient state graph data structure and the minimization algorithms we analyzed. For each graph problem, all algorithms are implemented in the same programming technology. This ensures that runtime measurements of both algorithms for a given problem are comparable. We measured runtime performance on an Intel® Core™ 2 Duo CPU E8400 3.0 GHZ machine with 3.25 GB RAM and Windows 7 (32 Bit). The energy saving

[2] http://www.cs.sunysb.edu/~algorith/implement/vflib/implement.shtml.

[3] http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/download/index.html.

[4] http://www.brics.dk/automaton/.

Table 3 Patterns

Collection		Type					Caption				
		Avg.	Std.	Min.	Max.	# Of patterns	Avg.	Std.	Min.	Max.	# Of patterns
SAP EPC	Nodes	4.30	1.23	1	7	399	6.82	4.11	1	17	2,023
	Edges	3.31	1.23	0	6		5.82	4.11	0	16	
PA1 EPC	Nodes	5.16	1.38	1	9	1,066	3.03	1.74	1	8	298
	Edges	4.17	1.38	0	8		2.03	1.74	0	7	
PA1 TTM	Nodes	3.00	1.58	1	5	5	1.28	0.72	1	5	176
	Edges	2.00	1.58	0	4		0.28	0.72	0	4	
PA1 ORG	Nodes	3.50	1.87	1	6	6	3.62	2.41	1	10	403
	Edges	2.50	1.87	0	5		2.62	2.41	0	9	
PA2 EPC	Nodes	11.20	2.84	1	20	28,184	4.98	2.62	1	10	363
	Edges	10.23	2.83	0	19		4.07	2.70	0	10	
PA2 ORG	Nodes	3.88	1.74	1	7	25	1.02	0.13	1	2	58
	Edges	2.88	1.74	0	6		0.02	0.13	0	1	
RI EPC	Nodes	12.16	2.97	1	21	27,580	2.18	1.31	1	7	283
	Edges	11.21	2.98	0	20		1.19	1.34	0	6	
RI ERM	Nodes	5.01	.41	1	7	223	2.97	.45	1	7	363
	Edges	8.06	.67	0	12		4.06	.69	0	12	
CWM CD	Nodes	6.25	.23	1	9	67	1.13	0.34	1	2	24
	Edges	11.01	.40	0	16		0.17	0.48	0	2	
MM ERM	Nodes	4.22	.24	1	9	37	3.17	.39	1	8	72
	Edges	6.43	.40	0	16		4.33	06	0	14	

mechanism was deactivated and the search process was executed as a real-time process. For gSpan and Gaston, we used the Oracle JVM 6.0.26 with a maximum heap size of 1.2 GB. The heap size is the limiting factor of our analysis because no heap size larger than 1.2 GB could be allocated by the 32 Bit JVM under Windows.

As we are interested in all pattern occurrences when searching for subgraphs, we configured Ullmann and VF2 correspondingly (otherwise, they return a Boolean "contains/not contains" value). We searched for each pattern in each model and calculated the total search time. gSpan and Gaston expect a support value as input parameter which determines the percentage of models containing a pattern. This means that a pattern is returned as soon as it occurs in at least X % of the models. We called both gSpan and Gaston with support values of 70, 50, 30, 20, 10, 5, and 1 %. The McGregor and Koch algorithms for maximum common subgraph detection were configured such that they return a maximum common subgraph that contains at least two adjacent nodes and the edge connecting them. Both algorithms furthermore require two input graphs. For each model collection, we randomly created 1,000 distinct model pairs and fed them to both the Koch and McGregor algorithms. If the collection was too small to create 1,000 pairs, all possible pairs were used. Furthermore, if no maximum common subgraph was found in a pair of

models within 3 min, we terminated the algorithms. All algorithms for state graph minimization did not require any further configuration.

4.5 Results for subgraph isomorphism

In Table 4, we provide results from our runtime analysis of VF2 and Ullmann. We conducted experiments for a number of different model collections, which can be found in the leftmost column. To each of them, we applied both VF2 and Ullmann to search for the corresponding patterns. To measure the effect of different label granularities, we conducted two separate experiments for each collection and algorithm. First, we considered only type information. Second, we considered also captions of model elements.

The runtimes we measured are search times for complete search runs, that is, searching for all occurrences of a single pattern in all the collection's models. This is necessary because runtimes for a single search, meaning searching occurrences of a single pattern graph in a single model graph, are too low to be reliably measured. A problem of this approach is that collection sizes vary a lot. Hence, they must be

Table 4 Normalized runtime measurements for subgraph isomorphism (Becker et al. 2012b)

Collection and algorithm		Type					Caption				
		Runtime (ms/model)				R^2	Runtime (ms/model)				R^2
		Avg.	Std.	Min.	Max.		Avg.	Std.	Min.	Max.	
SAP EPC	VF2	1.58	.91	1.33	19.46	.01	1.28	.10	1.12	2.78	.00
	Ull	1.54	.92	1.32	19.61	.00	1.23	.12	1.10	2.97	.00*
PA1 EPC	VF2	3.13	4.05	2.65	94.37	.00	2.72	.10	2.59	3.26	.00
	Ull	4.50	7.02	2.66	126.96	.02**	2.74	.15	2.58	3.66	.04**
PA1 TTM	VF2	14.70	28.55	1.11	65.73	.54	.97	.12	.90	1.36	.01
	Ull	16.11	31.93	1.05	73.18	.53	.98	.14	.90	1.71	.00
PA1 ORG	VF2	33.04	71.80	1.06	179.26	.48	.96	.35	.88	5.10	.00
	Ull	38.41	83.63	1.07	208.70	.48	.95	.28	.88	4.20	.00
PA2 EPC	VF2	1.69	.11	1.54	11.26	.00**	1.54	.07	1.46	2.15	.02*
	Ull	69.20	884.46	1.59	35,562.25	.02**	1.59	.09	1.47	2.37	.00
PA2 ORG	VF2	40.42	133.21	.94	598.11	.13*	1.10	.34	.94	2.44	.00
	Ull	53.19	185.70	1.00	875.56	.12*	.99	.08	.94	1.39	.04
RI EPC	VF2	1.62	.36	1.52	10.83	.00	1.40	.05	1.37	2.00	.03
	Ull	9.72	27.88	1.54	584.28	.09**	1.42	.20	1.37	4.81	.00
RI ERM	VF2	1.26	.10	.94	2.26	.01	1.16	.16	.94	3.33	.00*
	Ull	1.30	.10	.94	1.91	.00	1.17	.16	.96	5.12	.00
CWMCD	VF2	1.17	.05	.50	8.83	.02	1.17	.24	.97	2.00	.20*
	Ull	1.25	.06	1.00	9.90	.02	1.04	.09	.97	1.30	.04
MM ERM	VF2	1.73	.09	1.54	5.00	.38	1.65	.23	1.40	13.37	.01
	Ull	1.77	.09	1.60	2.26	.66**	1.49	.16	1.40	3.60	.00

normalized to make the numbers comparable. We divided the runtimes of search runs by the number of models in the corresponding collection. Table 4 reports summary statistics of normalized runtimes for these search runs.

To clarify this, consider the following example: Choosing collection SAP_EPC labeled only by model element types, and choosing VF2 as the algorithm, the average time needed to search one of the 399 patterns in all the 604 model graphs is 952.35 ms. Normalizing this number, we end up with $952.35/604 = 1.58$ ms, which is the average runtime reported in Table 4's column *Avg* in the *type* section. In a similar way, the other columns report the *Std.*, the *Min.*, and the *Max.* of normalized search times.

With the order of magnitude being milliseconds for almost all cases, we can see that both algorithms are well applicable to large model collections. With average normalized runtimes ranging from 1 to 70 ms, they can be expected to return pattern occurrences within seconds even if used on thousands of conceptual models. Looking at the average runtimes, VF2 appears superior to Ullmann, which is what we expected based on literature. In most cases, VF2 ran faster on average than Ullmann did. In those cases in which it did not, runtimes for both algorithms were very close. The biggest advantage of Ullmann over VF2 has been found when applying them to the CWM_CD labeled by types and captions. With runtimes of 1.17 ms for VF2 and 1.04 ms for Ullmann, VF2 was about 12.5 % slower than Ullmann. Often though, VF2 was much faster than Ullmann. For example, for collection PA2_EPC with considering only types, VF2 was about 40 times as fast as Ullmann.

Turning from average runtimes to the extreme cases, we see that Ullmann's runtime variability is much bigger than VF2's. The highest runtime for Ullmann has been observed in collection PA2_EPC (considering types only). In this case, there has been one pattern for which the normalized runtime amounts to 35,562.25 ms (about 35 s). Multiplying with the number of models in this collection, we see that the search run took almost 6 h to complete. For algorithm VF2, the highest runtime observed was 598.11 ms in collection PA2_ORG (considering types only). Hence, VF2 appears to be more reliable on conceptual models than Ullmann, which is another argument in favor of this algorithm.

Comparing search times for different label granularities, we see that when considering only types, runtimes are strictly higher as compared to considering captions. This is in line with what we expected theoretically. In most cases, the algorithm can immediately abort if it does not find a suitably labeled node in the model graph. We thus see that average runtimes never exceeded 3 ms, and that the maximum runtimes exceeded 10 ms only once. Searching patterns with VF2 in the MM_ERM collection (labeled by captions), one search run took 13.37 ms after normalization.

In summary, we can recommend using VF2 for SGI. It is faster on average than Ullmann in virtually all cases and there is less risk to encounter extreme runtimes. It appears that the feasibility rules VF2 uses to prune the search space are effective in avoiding unnecessary exploration. The Ullmann strategy of calculating all possible mappings first and filtering out those being isomorphic next was inferior on our collections.

In a next step, we tried to explain the runtime of search runs using properties of the algorithms' inputs. We used a linear regression model to regress both pattern node and edge counts on runtimes to see how much variance could be explained. Table 4's R^2 column reports on the coefficients of determination for each of the regressions.

Almost all coefficients in Table 4 are smaller than 0.05 and thus very low. Only in few cases, we have been able to explain more of the variance. However, most of these R^2 values are not significant. The main reason for insignificance is that, for some of the collections we analyzed, there have only been very few patterns to search for. Consider collection PA1_TTM labeled with captions as an extreme example. Searching for only 5 patterns can hardly deliver any significant statistical results. In some cases though, we have been able to discover significant influences. Most notably, we found an R^2 of 0.66 when regressing the results of applying Ullmann to collection MM_ERM labeled by captions. Positive results have also been found for the VF2 algorithm. Regressing the results of applying it to collection PA2_ORG labeled by types, we have been able to explain 13 % of runtime variance. Unfortunately, positive results are the exception rather than the rule. In general, we have not been successful in predicting runtimes of search runs for patterns in collections. Therefore, we deviated from our method and skipped the overfitting test. These results contradict previous research about these algorithms to some extent (Cordella et al. 2004).

Furthermore, we tried to explain the runtime for VF2 and Ullmann after combining the results for all collections. As explaining variables, we chose the average node and edge counts of a collection's models, their interaction (the product), as well as node and edge counts of a pattern and their interaction. The explained variable was the normalized runtime. In Table 5, we provide the R^2's of this analysis. In the case of available captions, the linear regression model works quite well as a runtime predictor. For both algorithms we could explain 95 % of runtime variance with a high significance. Thus, the explaining variables can predict runtime performance reasonably well. If only the type information is used, we could explain only 35 % of runtime variance for VF2 and 0 % for the Ullmann algorithm.

Comparing R^2_{Train} and R^2_{Test}, we do not see serious drops in variance explained when applying the estimator to the test data. Hence, it does not overfit and can thus be used to estimate runtime performance.

Given that we found both algorithms performing well in almost all cases, we decided to visually inspect the models involved in search runs taking more than 1,000 ms. There were 17 such search runs, all involving the Ullmann algorithm, and the corresponding patterns were relatively large (13–18 nodes and 12–17 edges). This may indicate that huge runtimes occur only when large patterns are searched. However, there have been bigger patterns used with this collection (maximum of 19 nodes and 20 edges, cf. Table 3), which did not take that much time. Hence, large patterns appear to be necessary but not sufficient for large runtimes. Inspecting the 17 patterns visually, we have not been able to find any notable properties that could be used to predict upfront if searching them in a collection will take long or not.

Table 5 Linear regression over all collections for subgraph isomorphism

Collection and Algorithm	Type			Caption		
	R^2	R^2_{Train}	R^2_{Test}	R^2	R^2_{Train}	R^2_{Test}
All						
VF2	.35**	.32	.72	.95**	.95	.95
Ull	.00**	.00	.00	.95**	.95	.94

Given that Ullmann permutes the adjacency matrix in an arbitrary way, an explanation could be that, by chance, a bad permutation was chosen in these cases.

4.6 Results for frequent pattern detection

In Tables 6 and 7, the average runtimes are listed in milliseconds (Avg.), as well as the number of identified frequent patterns (#p) related to the level of support. Table 6 contains the results for labeling nodes with types and captions, whereas Table 7 lists the results for labeling nodes with types only. As memory requirements are an important issue in frequent pattern detection, we observed some out of memory errors, indicated by "—".

Some support levels were not applicable to small model collections. For example, PA2_ORG contains only 18 models, so applying a support of 5 % would require occurrences of a pattern in a minimum of 0.9 models. In consequence, frequent pattern detection would return all possible subgraphs of any model. Hence, we required a pattern to occur in at least two models and tagged all inapplicable cases with "#M $<$ 2". Note that gSpan and Gaston return different amounts of patterns in some cases. The reason is that Gaston treats directed graphs as if they were undirected while gSpan also considers directed edges. Thus, the numbers of patterns are only the same for the ERM and UML class diagram collections.

Most runtimes reported in the tables are below 1 s, except two cases. Gaston needed 78 s for MM_ERM with 5 % support, gSpan took 26 s. If we labeled nodes only with types, performance decreases significantly (cf. Table 7). We expected this decrease, since, due to the higher number of mapping possibilities, the number of frequent patterns increases. However, runtimes remain below 10 s up to a support of 10 %. Smaller supports clearly increase runtimes. It took gSpan, for example, more than 150,000 ms ($\sim$2.5 min) to process RI_ERM with a support of 5 %.

In most cases, gSpan performed better than Gaston, which contradicts previous empirical studies (Marc et al. 2005; Nijssen and Kok 2006). Obviously, the lower performance of Gaston cannot be explained with its inability to process directed edges. For example, if we apply both algorithms to undirected graphs, gSpan still performs better (cf., e.g., MM_ERM with 5 % support; here, gSpan is about three times as fast as Gaston). The reason why Gaston was considered to be fast in previous studies was that it saves preliminary mappings, while gSpan recalculates them every time. We explain the lower performance with the structure of conceptual models, which are mostly sparse. Hence, it might be inefficient to save and augment partial mappings in comparison to recalculating them.

Table 6 Runtime measurement for frequent subgraph detection (captions) (Becker et al. 2012b)

Collection and algorithm		70 %		50 %		30 %		20 %		10 %		5 %		1 %	
		Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p
SAP EPC	Gaston	31	0	42	2	42	4	41	5	43	6	50	15	1,033	2,042
	gSpan	41	0	41	2	38	3	47	5	51	8	56	14	443	2,023
PA1 EPC	Gaston	144	2	157	3	188	4	240	7	358	13	691	105	—	—
	gSpan	149	2	152	3	185	4	239	7	350	15	587	87	—	—
PA1 TTM	Gaston	16	0	24	0	16	0	16	0	16	0	16	0	47	17
	gSpan	16	0	15	0	24	0	16	0	16	0	24	0	31	17
PA1 ORG	Gaston	16	0	16	0	16	0	15	0	16	0	15	0	#M < 2	#M < 2
	gSpan	8	0	16	0	16	0	8	0	16	0	8	0	#M < 2	#M < 2
PA2 EPC	Gaston	44	1	42	1	48	3	53	5	58	5	75	11	10,375	30,948
	gSpan	45	1	46	1	46	3	55	4	58	5	73	12	5,428	30,337
PA2 ORG	Gaston	16	0	16	0	8	0	16	2	109	58	#M < 2	#M < 2	#M < 2	#M < 2
	gSpan	8	0	15	0	8	0	8	2	16	58	#M < 2	#M < 2	#M < 2	#M < 2
RI EPC	Gaston	27	2	22	3	26	5	25	6	27	11	102	82	#M < 2	#M < 2
	gSpan	19	2	20	2	24	5	26	7	27	11	60	76	#M < 2	#M < 2
RI ERM	Gaston	15	0	20	4	29	7	31	14	67	43	188	435	#M < 2	#M < 2
	gSpan	12	0	16	4	16	7	31	14	24	43	172	470	#M < 2	#M < 2
CWM CD	Gaston	10	0	10	0	11	0	16	2	25	12	53	22	#M < 2	#M < 2
	gSpan	9	0	9	0	9	0	11	2	12	12	16	22	#M < 2	#M < 2
MM ERM	Gaston	23	3	23	7	31	17	47	36	219	810	78,391	360,334	#M < 2	#M < 2
	gSpan	24	3	32	7	40	17	55	36	172	810	26,154	360,334	#M < 2	#M < 2

Unlike our other experiments, we cannot construct a statistical runtime estimator for frequent pattern detection. The reason is that the atomic operation in this experiment is the search for frequent patterns in a model collection. Hence, we have only few data points, which is why a statistical analysis would not be meaningful. There is no worst-case runtime complexity known for both algorithms to the best of our knowledge. Hence, we could not compare the results to the worst-case complexity.

Both gSpan and Gaston sometimes aborted due to memory overflow (indicated by "—"). Again, gSpan outperforms Gaston concerning memory requirements. Hence, for analyzing conceptual models, gSpan is preferable with respect to memory requirements. We expected this result due to the buffering strategy of Gaston.

However, if many frequent patterns are found, both gSpan and Gaston suffer from memory overflow. If we use both captions and types as node labels, we observe that memory exceptions almost never occur (cf. Table 6). The reason is that, due to detailed label information, we will only find few frequent patterns, even for low values of support. Labeling nodes only with types leads to finding many frequent patterns, even for high levels of support (cf. Table 7). For example, the algorithms return more than 400,000 patterns at a support level of 5 % (i.e., the pattern has to occur in two or more models) when applied to the 35 models of MM_ERM. For larger collections, such as PA1_EPC containing 2,200 models, gSpan and Gaston fail quickly.

Applying the algorithms to organizational charts with nodes labeled by types produces notable results (cf. Table 7). High support values (20 % for PA1_ORG and 10 % for PA2_ORG) caused even gSpan to fail due to memory exceptions. This was in spite of very few patterns found on the last applicable support level (max. 20). Also, PA1_ORG and PA2_ORG are comparatively small (88 and 18 models). Organizational charts seem to be particularly challenging for gSpan and Gaston, which is surprising as organizational charts are mostly trees and hence have a simple structure.

Figure 2 provides a more detailed analysis of the memory requirements of both gSpan and Gaston. All diagrams depicted in the figure contain the support level on the abscissa and the number of megabytes required to identify frequently occurring patterns within the respective model repository on the ordinate. Missing values indicate that memory requirements exceeded 3.25 GB of RAM. Memory requirements of the Gaston algorithm fall within a range of about 15–1,000 MBs depending on the level of support. The memory requirements of gSpan on the other hand reach a maximum value of about 600 MBs. The figure demonstrates that the memory requirements of the Gaston algorithm are strictly greater than those of gSpan. This is due to the aforementioned fact that Gaston stores previously calculated mappings. The difference between the two algorithms is particularly huge in case of patterns that contain element types as labels. In this case, a great number of frequent patterns are found which strongly increases the memory requirements of Gaston, whereas gSpan only updates the appearance lists, which take up considerably less memory space. Furthermore, the figure shows how memory requirements increase with decreasing level of support. This holds true for

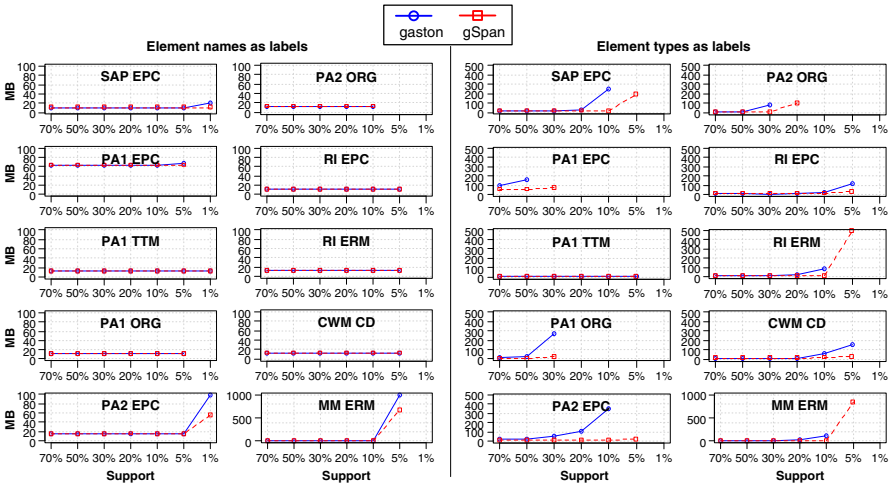

Fig. 2 Memory requirements of gSpan and Gaston

both algorithms, because a decreasing level of support leads to a greater number of frequent patterns. The PA1 model collections containing EPCs and organizational charts are particularly difficult for both algorithms to analyze. They both fail at rather high levels of support in case model elements are labeled with their element types only.

4.7 Results for maximum common subgraph detection

Table 8 contains runtime measurements in milliseconds for the two algorithms for maximum common subgraph detection by Koch and McGregor. The Avg. column contains the mean runtime it took the respective algorithm to search for the maximum common subgraphs in one pair of models contained in a given collection. The Min. and Max columns contain the overall minimum and maximum runtime. The Std. column reports on the Std. Note that we configured both algorithms in such way as to only return common subgraphs that have at least two adjacent nodes and the edge connecting them (cf. Sect. 4.4). Furthermore, two nodes can only be mapped to one another if they share the same type and caption (cf. Sect. 4.1).

Preliminary experiments we conducted show that the maximum runtimes for the Koch and McGregor algorithms may be very large. We encountered pairs of conceptual models for which these algorithms did not return a result even after 2 h. Hence, we decided to work with a maximum runtime in our experiments. For any pair of models, we aborted the execution once a runtime of 3 min was exceeded. To do so, we added a checkpoint to the algorithms' source codes. Consequently, the maximum runtimes reported in Table 8 slightly exceed 3 min, as the checkpoint is reached only every few seconds. In case of a timeout, the maximum runtime of 3 min was included in the average runtime calculation for a model pair. We do this because there are many cases in which one algorithm times out while the other does

Table 7 Runtime Measurement for Frequent Subgraph Detection (types) (Becker et al. 2012b)

Collection and algorithm		70 %		50 %		30 %		20 %		10 %		5 %	
		Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p	Avg.	#p
SAP EPC	Gaston	58	3	100	13	149	30	303	52	2,027	144	—	—
	gSpan	72	4	75	7	123	30	150	42	1,137	137	11,710	399
PA1 EPC	Gaston	655	13	1,783	42	—	—	—	—	—	—	—	—
	gSpan	506	8	1,405	40	19,410	275	—	—	—	—	—	—
PA1 TTM	Gaston	3,534	4	—	—	—	—	—	—	—	—	—	—
	gSpan	2,262	4	61,713	5	—	—	—	—	—	—	—	—
PA1 ORG	Gaston	63	3	141	4	1,568	5	—	—	—	—	—	—
	gSpan	71	3	125	4	1,139	5	—	—	—	—	—	—
PA2 EPC	Gaston	245	18	410	36	894	154	2,064	423	6,190	1,945	—	—
	gSpan	160	17	283	41	672	146	1,519	401	4,796	1,733	16,464	8,467
PA2 ORG	Gaston	23	3	24	4	413	6	—	—	—	—	#M < 2	#M < 2
	gSpan	16	2	24	4	390	6	5,733	20	—	—	#M < 2	#M < 2
RI EPC	Gaston	55	35	76	79	132	310	210	849	506	4,004	2,795	33,868
	gSpan	67	39	85	93	148	318	242	780	591	3,570	2,479	27,580
RI ERM	Gaston	39	12	47	36	125	165	343	555	1,731	4,688	—	—
	gSpan	39	12	55	36	141	168	390	574	1,965	4,816	150,861	315,463
CWM CD	Gaston	33	3	38	5	76	10	93	15	596	66	1,725	190
	gSpan	26	2	38	5	42	6	98	12	859	107	2,592	364
MM ERM	Gaston	55	14	62	37	125	160	328	462	1,467	3,589	—	—
	gSpan	71	14	55	37	141	160	429	462	1,981	3,589	133,263	402,190

Table 8 Runtime measurements for maximum common subgraph detection

Collection and algorithm		Runtime (ms)				# Timeout	# Of Wins	R^2	R^2_{Train}	R^2_{Test}	Cumulated runtime (ms)
		Avg.	Std.	Min.	Max.						
SAP EPC	Koch	186.06	5,715.44	0.01	180,780.44	1	999	0.73**	0.71	0.85	186,061.48
	McGregor	8,143.88	36,546.34	0.04	180,004.13	42	1	0.48**	0.47	0.52	8,143,875.30
PA1 EPC	Koch	2,011.22	18,919.90	0.01	189,544.11	11	985	0.61**	0.63	0.56	2,011,220.30
	McGregor	26,761.93	63,522.82	0.04	180,015.46	146	15	0.35**	0.35	0.38	26,761,926.36
PA1 TTM	Koch	0.04	0.19	0.01	4.77	0	993	0.34**	0.34	0.40	37.54
	McGregor	0.10	0.16	0.03	2.57	0	7	0.67**	0.68	0.69	101.90
PA1 ORG	Koch	0.07	1.45	0.01	45.79	0	992	0.38**	0.36	0.55	71.63
	McGregor	0.10	0.29	0.02	4.90	0	8	0.73**	0.75	0.67	104.24
PA2 EPC	Koch	09	18,041.44	0.01	181,064.46	10	990	0.53**	0.59	0.33	1,888,090.75
	McGregor	6.83	60,733.09	0.06	180,097.67	130	10	0.33**	0.34	0.31	24,476,829.04
PA2 ORG	Koch	0.03	0.06	0.01	0.55	0	1	0.79**	0.82	0.69	5.28
	McGregor	0.20	0.67	0.03	8.11	0	0	0.96**	0.97	0.96	30.50
RI EPC	Koch	0.29	0.77	0.02	20.93	0	1,000	0.81**	0.82	0.81	289.63
	McGregor	5.37	56,325.70	0.10	180,005.41	107	0	0.40**	0.41	0.42	20,185,371.31
RI ERM	Koch	4.89	44.61	0.07	688.09	0	519	0.18**	0.22	0.09	2,582.11
	McGregor	1,324.79	14,658.75	0.13	180,006.41	3	9	0.42**	0.40	0.52	699,486.67
CWM CD	Koch	0.04	0.03	0.01	0.23	0	1	0.76**	0.77	0.73	15.42
	McGregor	0.08	0.06	0.03	0.62	0	0	0.78**	0.79	0.80	36.50
MM ERM	Koch	28,721.96	63,609.03	0.21	187,701.60	81	474	0.06**	0.06	0.07	17,089,565.28
	McGregor	51,547.09	80,056.66	0.46	180,005.10	162	121	0.001*	0.00	0.02	30,670,519.49
All	Koch	2,745.04	21,492.93	0.01	189,544.11	103	6,954	0.37**	0.38	0.36	21,177,939.42
	McGregor	14,402.89	48,250.93	0.02	180,097.67	590	171	0.47**	0.46	0.50	110,938,281.31

not. Thus excluding these values would render our summary statistics incomparable. The timeout column reports on the number of timeouts that occurred given a particular algorithm and given model collection.

The second to last column on the right-hand side of Table 8 (# of wins) reports on which algorithm was able to return results faster. Consider, for example, the SAP_EPC collection. In 999 out of 1,000 cases, the Koch algorithm was able to return results faster than the McGregor algorithm. The last column of Table 8 contains the cumulated runtime for all model pairs.

Table 8 demonstrates that the Koch algorithm significantly outperforms the McGregor algorithm. In all model collections the Koch algorithm is able to return results faster than McGregor. This is supported by both the average as well as the cumulated runtime measurements. Koch also runs into less timeouts. Consider, for example, the RI_EPC collection. The McGregor algorithm runs into 107 timeouts. McGregor seems to be significantly slower on EPC models. Again, the RI_EPC collection clearly demonstrates this finding, as it took the Koch algorithm an average of 0.29 ms to find a maximum common subgraph, whereas it took the McGregor algorithm about 20 s. Furthermore, the Std. of the Koch algorithm was in most cases much lower than the Std. of McGregor. The Koch strategy of calculating the vertex product graph first and then searching for the maximum clique in this graph is significantly faster on conceptual models than the backtracking strategy of McGregor. We therefore generally recommend the Koch algorithm, as it exhibits both faster and more stable runtime behavior.

In order to find an estimator for the runtime of the maximum common subgraph algorithms, we tried a multivariate regression model with the nodes of two graphs (N_1; N_2) as independent variables and the runtime as the dependent variable. Furthermore, we added the interaction ($N_1 * N_2$) as the independent variables are not independent among *themselves*. The bigger one of the input graphs is the more checks between the nodes have to be computed (Levi 1973). An investigation of the runtime distribution for each collection revealed that some were skewed due to outliers. Therefore, we conducted a logarithmic transformation of the runtime. Using this linear regression model for the McGregor algorithm over all collections results in $R^2 = 0.47$.

The resulting R^2 for each collection are depicted in column R^2 in Table 8. The best R^2 of 0.81 for the Koch algorithm was found in the RI_EPC collection. For the McGregor algorithm the best R^2 value of 0.96 was found in the PA2_ORG collection. The lowest R^2 values of 0.06 and 0.001 are in the MM_ERM collection. The collection has 35 nodes and 72 edges on average, leading to the highest average edge count per node of 2 over all collections. This high amount of edges seems to be leading to a non-linear increase in states to be checked and therefore in high non-linear execution times. The R^2 over all collections are presented at the bottom of Table 8. For the McGregor algorithm 47 % and for Koch 37 % of variation can be explained with the linear regression model. As the R^2_{Train} values are not much higher than R^2_{Test} values, no overfitting of the estimator took place.

Within the collection CWM_CD and PA2_ORG the runtime is somewhat predictable for both algorithms ($R^2_{Test} > 0.69$). In the other collections, the estimator

Table 9 Runtime measurements for state graph minimization

k	Collection	Models used	Size (states + transitions)				Avg. size reduction (%)	Algorithm	Time (ms)						
			Avg.	Std.	Min.	Max.			Avg.	Std.	Min.	Max.	R^2	R^2_{Train}	R^2_{Test}
1	BIT_A PN	279	167.43	595.27	9	6,760	9.13	Hu	30.48	307.10	0	3,947	.98**	0.99	0.99
								Br	15.48	101.74	0	1,538	.80**	0.81	0.81
								Ho	5.73	29.18	0	408	.88**	0.89	0.66
	BIT_B3 PN	313	675.58	1,493.19	2	1,923	4.52	Hu	118.61	429.09	0	3,374	.96**	0.96	0.99
								Br	30.91	89.12	0	970	.78**	0.79	0.69
								Ho	22.17	102.73	0	1,574	.61**	0.60	0.64
	BIT_C PN	23	286.65	652.74	23	2,672	13.62	Hu	31.74	126.05	0	602	.89**	0.89	0.95
								Br	28.30	94.33	0	457	.74**	0.73	0.93
								Ho	24.04	90.12	0	436	.69**	0.69	0.89
2	BIT_A PN	275	165.58	428.55	9	4,861	14.55	Hu	15.04	110.59	0	1,627	.96**	0.96	0.99
								Br	13.24	61.62	0	678	.58**	0.60	0.89
								Ho	4.61	14.63	0	136	.83**	0.82	0.95
	BIT_B3 PN	307	724.45	1,648.44	3	9,467	5.54	Hu	137.18	511.22	0	3,988	.88**	0.88	0.89
								Br	31.90	91.44	0	970	.70**	0.71	0.76
								Ho	23.23	105.79	0	1,574	.40**	0.68	0.96
	BIT_C PN	23	286.65	652.74	23	2,672	13.62	Hu	31.35	124.27	0	593	.90**	0.99	0.99
								Br	28.30	94.55	0	458	.74**	0.73	0.93
								Ho	23.57	88.67	0	429	.69**	0.99	0.78

Table 9 continued

k	Collection	Models used	Size (states + transitions)				Avg. size reduction (%)	Algorithm	Time (ms)						
			Avg.	Std.	Min.	Max.			Avg.	Std.	Min.	Max.	R^2	R^2_{Train}	R^2_{Test}
3	BIT_A PN	272	210.47	657.25	9	5,418	18.08	Hu	40.91	295.82	0	3,017	.90**	0.90	0.92
								Br	25.90	178.55	0	1,987	.68**	0.55	0.82
								Ho	7.22	37.34	0	503	.73**	0.73	0.91
	BIT_B3 PN	305	684.72	1,586.86	3	11,952	5.22	Hu	130.56	492.53	0	3,984	.74**	0.71	0.97
								Br	29.46	77.40	0	525	.74**	0.73	0.81
								Ho	18.22	56.20	0	537	.57**	0.60	0.63
	BIT_C PN	23	286.65	652.74	23	2,672	13.62	Hu	31.39	125.71	0	601	.89**	0.88	0.99
								Br	28.87	96.37	0	467	.73**	0.73	0.93
								Ho	23.57	88.90	0	430	.66**	0.95	0.65

works for one algorithm but not the other (SAP_EPC, RI_EPC) or not at all (MM_ERM). Hence, the runtimes of maximum common subgraph algorithms appear not to be well predictable on process model collections with a linear regression model. However, it may serve as a starting point for a further investigation.

4.8 Results for state graph minimization

Summarized results for our experiments with state graph minimization can be found in Table 9. It lists runtimes for comparison of all three algorithms under evaluation. Remember that state graphs have been obtained from Petri nets under the assumption of k-boundedness for varying k. Table 9 is subdivided into three parts corresponding to the experiments with $k = 1$, $k = 2$, and $k = 3$, as indicated by the leftmost column. Right beside this column, the reader can find the collection used. Please note that, in contrast to the other experiments, we did not apply the state graph algorithms to the original conceptual models, i.e., the petri nets. Rather, we derived the state graphs from the petri nets.

The column "Models used" lists the number of models that could be used for the experiment. As state spaces can grow enormously, some of the more complex models would have produced out of memory exceptions. The authors in (Fahland et al. 2009) analyzed the state spaces of these models before and found that their size can easily exceed 1,000,000 states. To avoid out of memory errors that would crash the JVM, we modified the code of the reachability graph generation algorithm and the three state graph minimization algorithms such that memory usage could be monitored. Whenever the JVM used more than 80 % of available memory, we skipped the model, called the garbage collector, and moved on to the next. The column "Models used" indicates the number of models that could be processed without running low on free memory.

The subsequent columns list summary statistics for the state graphs in each collection. They indicate Avg., Std., Min. and Max. of the state graphs' sizes. Size has been measured as the sum of the number of states and the number of transitions in that graph. As it can be seen, the average size does not exceed 725, and the maximum size is 11,952. These numbers are orders of magnitude away from 1,000,000 or even more, which would have been returned with unlimited memory. On the one hand, this is not particularly surprising as no one would expect to be able to process very huge state spaces on personal computers. What we think is surprising though is that a huge number of processes has rather small state spaces. This is most evident in collection BIT_A_PN, of which only 10 models had to be skipped due to memory requirements, and in which the average size over the included state graphs is only about 210. In the other collections, however, a substantial number of models were skipped. Hence, these numbers will contain a bias that cannot be neglected and should be interpreted with care.

In column "Avg. size reduction," we listed by how much a state graph can be reduced if a minimization algorithm is applied. It quantifies the degree to which models in the corresponding collection could be simplified. Fractions range from 4.52 up to 18.08 %. Hence, there is much variability between the different

collections. However, these results show that there is opportunity for significant reduction of complexity in real-world models. For algorithms with performance being highly dependent on the state graph size, a reduction of almost 20 % could improve performance a lot.

The column "Algorithm" indicates to which minimization algorithm the runtime statistics of the corresponding row belongs to (**Hu**ffman, **Br**zozowski, or **Ho**pcroft). These runtime statistics can be found in the right part of Table 9 and again, Avg., Std., Min. and Max. runtimes in milliseconds are listed. Comparing the different algorithms, we find that the Huffman algorithm is, on average, slower than the competitors regardless of the collection and the assumption on boundedness. Its average runtime is worst for collection BIT_B3_PN under 2-boundedness (137.18 ms) and maximum runtime was up to about 4 s in several cases. Thus, our experiments suggest this algorithm should not be applied to business process models.

Second best in terms of average runtime is the Brzozowski algorithm. Its average runtime is always smaller than that of the Huffman algorithm though always larger than that of the Hopcroft algorithm and ranges between 13 and 32 ms. This may suggest to give a general recommendation towards the Hopcroft algorithm, as it can be expected to deliver results faster when applied to a large number of models. Its average runtimes vary between 4.5 and 24 ms. Turning to the Std., we found that that of the Brzozowski algorithm was smaller than that of the Hopcroft algorithm in some cases. This was the case for BIT_B3_PN and $k \neq 3$. However, Stds were only slightly larger and increased from 89.12 to 102.73 for $k = 1$ and from 91.44 to 105.79 for $k = 2$. In other cases, the Std. of the Hopcroft algorithm was much lower than that of the Brzozowski algorithm. Consider BIT_A_PN and $k = 1$, for which the former produced a Std. of only 29.18 whereas the latter produced one of 101.74. Similarly, maximum runtimes of the Brzozowski algorithm are most often much larger than that of Hopcroft (1,987 for Brzozowski vs. 970 for Hopcroft). Consequently, the Hopcroft algorithm appears to exhibit both faster and more stable runtime behavior as compared to the Brzozowski algorithm. Hence, we generally recommend using this algorithm regardless of whether average runtime or exceptionally long runtime is to be minimized.

To predict the runtimes of these algorithms based on the input models, we used a regression model with the size of the state graph (sum of states and transitions) as the independent variable. For the Huffmann algorithm, scatter plots suggested quadratic growth of runtime with size, while for the other algorithms, a linear predictor seems to work best. The percentage of variance explained by these predictors can be found in the rightmost column. Predictions are most accurate for the Huffman algorithm (R^2 ranging from .75 up to .98). However, also the runtimes of the Brzozoswki and Hopcroft algorithms can be explained well in many cases. Hence, the runtimes of state graph minimization algorithms appear to be well predictable on process model collections. Comparing R^2_{Train} and R^2_{Test}, we only see an overfitting for the BIT_C_PN collection. However, this collection includes only 23 models, so the test set encompasses only four runtime measurements and is therefore error prone to outliers. For the other collections, no overfitting took place.

5 Discussion

Concerning different types of models, we expected differences in the performance of the algorithms as they use different strategies to build up and parse the search space. Indeed, different types of models did produce different results (e.g., the memory problems of Gaston and gSpan when analyzing organizational charts). However, we could not observe that one algorithm is better than another depending on specific types of models. Rather, our results indicate that certain algorithms are preferable in almost all cases.

Due to the results of the performance evaluation, we expect the algorithms to perform well in a graph-based analysis framework as proposed in Sect. 2. However, reusing basic graph algorithms requires taking some special characteristics of model analysis into account. On the one hand, nodes and edges of conceptual model graphs are labeled, with possibly complex data within these labels. Moreover, many conceptual models have directed and undirected edges at the same time. On the other hand, basic graph algorithms mostly (not always) neglect labels, or only consider either directed or undirected edges. Therefore, implementing basic graph algorithms in a graph-based analysis framework requires extending them so that they can handle any type of edge as well as complex, multivariate node and edge labels. Furthermore, especially algorithms for SGI, maximum common subgraph detection and frequent pattern detection incorporate label and edge mappings. Consequently, they require a notion of node or edge similarity. Hence, we must be able to compare nodes and edges not only based on their structural environment, but also based on their label values. Depending on the analysis purpose, equality checks have to be configurable, regarding, for example, which values of the multivariate label should be compared and what equivalence relation should be used [e.g., composed of Levenshtein distance (Levenshtein 1966), numerical comparison, ontological similarity (Thomas and Fellmann 2009), linguistic similarity (Delfmann et al. 2009), combinations, etc.].

Additional semantic checks will certainly have an influence on runtime performance. However, it is difficult to predict whether it will speed up the algorithms or slow them down. As we demonstrate in the performance evaluation, considering additional label information speeds up the processing speed because it reduces the search space. Considering additional information can also slow down the processing speed due to additional checks. This can be particularly true when it comes to non-trivial similarity checks like linguistic or ontological equality checks. In any case, indexing techniques can be used to prune the search space and speed up the matching process (Jin et al. 2010). The runtime performance of high-level model analysis approaches developed with the framework will thus differ from the measurements of the low-level graph algorithms presented in this paper.

Furthermore, the overall runtime performance of a high level model analysis approach also incorporates the overhead for pre- and post-processing steps. These could be, for example, loading and filtering models or further processing the results returned by the graph algorithms. As implementations for many graph algorithms are already available, developing the application logic for these pre- and post-processing steps will be the main challenge. In particular, this requires further

analyzing model management and analysis tasks in order to derive corresponding requirements.

6 Summary and outlook

In this paper, we introduced an architecture of a graph algorithm-based model management and analysis framework supporting conceptual model analysis tasks. We furthermore conducted a runtime analysis of various graph algorithms. The paper demonstrates that, despite their theoretical intractability, these algorithms perform well on conceptual models in many cases and can thus be used to query even large collections of conceptual models. In terms of *SGI*, the VF2 algorithm outperforms Ullmann in many cases. As far as algorithms for *frequent pattern detection* are concerned, gSpan outperforms Gaston in both runtime and memory requirements. In terms of *maximum common subgraph detection*, our analysis suggests that the Koch algorithm is significantly faster than the McGregor approach. In terms of algorithms designed for a *behavioral model analysis*, the Hopcroft algorithm provides the best runtime performance.

A limitation of our study is the fact that we only considered models created in a limited set of modeling languages. Although we cover various types of conceptual models ranging from process models to data models, organizational charts, and TTM, runtime experiments should be repeated on models created in different modeling languages in order to confirm and extend the results. Furthermore, to be able to use the algorithms we propose as part of the framework, the algorithms need to be extended to deal with the particular requirements of analyzing conceptual models (cf. Sect. 5).

The algorithms for SGI and frequent pattern detection presented in this paper return exact matches. In many model analysis scenarios, however, we are more interested in finding structures that are similar to a given query (Wang et al. 2013). In graph theory, this is known as the problem of subgraph homeomorphism. The framework as we envision therefore should also include algorithms for this problem as well. At the moment, we are working on a particular algorithm on subgraph homeomorphism for conceptual models by extending VF2. Also frequent subgraph miners need to be augmented to allow for mining frequently occurring structures that are not strictly isomorphic to one another, but similar (Ekanayake et al. 2012).

Another limitation is that our choice of state graphs as a representation for dynamic behavioral aspects of processes is somewhat arbitrary. While state graphs are popular, other representations exist which could also be subject to minimization. We leave it to future research to investigate these issues.

Short to midterm research will focus on developing and implementing the framework proposed in Sect. 2. We have already started to do so. So far, the algorithm layer includes implementations of the Ullmann and VF2 algorithms. Implementations for the frequent pattern miner gSpan and the maximum common subgraph miner by Koch are underway. Concerning the runtime changes of algorithms when considering both directed and undirected edges, multivariate labels for both nodes and edges and arbitrary equivalence relations for the comparison of

nodes and edges, we plan to perform further performance analyses, as soon as the changes are implemented.

Long term research will focus on extending the framework to include high-level model analysis approaches proposed in the literature. First starting points are the approaches presented in Sect. 3.1. Their sub-procedures to search for maximum common regions (La Rosa et al. 2013) or frequent clones (Dumas et al. 2013) can then be replaced by calls to the respective graph algorithms. Runtime experiments need to be performed on the high-level analysis approaches to determine their practical applicability.

Despite the abundance of analysis approaches proposed in the literature, hardly any of them have currently disseminated into practice, despite evidence that companies have started to create and maintain large collections of conceptual models. We believe this due to the highly specialized nature of proposed approaches that are often designed to address one particular model management task alone and may be restricted to analyzing models of a specific type or modeling language. In our opinion, using a framework as proposed above is a remedy for that and may eventually lead to model analysis approaches disseminating into corporate reality.

References

Aalst WMP, Hee KM, Hofstede AHM, Sidorova N, Verbeek HMW, Voorhoeve M, Wynn MT (2010) Soundness of workflow nets: classification, decidability, and analysis. Form Asp Comput 23:333–363

Awad A (2007) BPMN-Q: a language to query business processes. In: Reichert M, Strecker S, Turowski K (eds) In: Proceedings of the 2nd international workshop on enterprise modelling and information systems architectures (EMISA'07). St. Goar, pp 115–128

Awad A, Sakr S (2010) Querying graph-based repositories of business process models. In: Yoshikawa M, Meng X, Yumoto T, Ma Q, Sun L, Watanabe C (eds) Proceedings of the 15th international workshops on database systems for advanced applications. Springer, Berlin, pp 33–44

Awad A, Polyvyanyy A, Weske M (2008) Semantic querying of business process models. In: 12th international IEEE enterprise distributed object computing conference (EDOC 2008). IEEE, Munich, pp 85–94

Awad A, Decker G, Weske M (2008) Efficient compliance checking using BPMN-Q and temporal logic. In: Dumas M, Reichert M, Shan M-C (eds) Proceedings of the 6th international conference on business process management (BPM'08). Springer, Berlin, pp 326–341

Batra D (2005) Conceptual data modeling patterns. J Database Manag 16:84–106

Becker J, Schütte R (2004) Handelsinformationssysteme. Redline Wirtschaft, Frankfurt

Becker J, Breuker D, Weiß B, Winkelmann A (2010a) Exploring the status quo of business process modelling languages in the banking sector—an empirical insight into the usage of methods in banks. 21st Australasian Conference on Information Systems (ACIS 2010). Brisbane, Australia

Becker J, Bergener P, Räckers M, Weiß B, Winkelmann A (2010b) Pattern-based semi-automatic analysis of weaknesses in semantic business process models in the banking sector. Proceedings of the European conference on information systems (ECIS'10). Pretoria, South Africa

Becker J, Breuker D, Delfmann P, Dietrich H-A, Steinhorst M (2012a) Identifying business process activity mappings by optimizing behavioral similarity. In: Proceedings of the 18th Americas conference on information systems (AMCIS 2012). Seattle, Washington

Becker J, Breuker D, Delfmann P, Dietrich H-A, Steinhorst M (2012b) A runtime analysis of graph-theoretical algorithms to detect patterns in process model collections. In: La Rosa M, Soffer P (eds) Proceedings of the 2nd International Workshop on Process Model Collections. Tallinn, Estonia, pp 31–42

Beeri C, Eyal A, Kamenkovich S, Milo T (2005) Querying business processes with BP-QL. In: Proceedings of the 31st international conference on very large data bases (VLDB'05). VLDB Endowment, Trondheim, pp 1255–1258

Beeri C, Eyal A, Kamenkovich S, Milo T (2006) Querying business processes. In: Dayal U, Whang K-Y, Lomet D, Alonso G, Lohman G, Kersten M, Cha SK, Kim Y-K (eds) Proceedings of the 32nd international conference on very large data bases (VLDB'06). VLDB Endowment, Seoul, pp 343–354

Beeri C, Eyal A, Kamenkovich S, Milo T (2008) Querying business processes with BP-QL. Inf Syst 33:477–507

Bräuer S, Delfmann P, Dietrich H-A, Steinhorst M (2013) Using a generic model query approach to allow for process model compliance checking—an algorithmic perspective. In: Alt R, Franczyk B (eds) In: Proceedings of the 11th International Conference on Wirtschaftsinformatik (WI) 2013. Universität Leipzig, Leipzig, pp 1245–1259

Brzozowski JA (1962) Canonical regular expressions and minimal state graphs for definite events. Symposium on the mathematical theory of automata. New York, pp 529–561

Cordella LP, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans Pattern Anal Mach Intell 26:1367–1372

Curran TA, Keller G (1998) SAP R/3 business blueprint: business engineering mit den R/3-referenzprozessen. Addison-Wesley, Bonn

Delfmann P, Herwig S, Lis L (2009) Unified enterprise knowledge representation with conceptual models—capturing corporate language in naming conventions. 30th International conference on information systems (ICIS 2009). Phoenix, Arizona

Derguech W, Vulcu G, Bhiri S (2010) An indexing structure for maintaining configurable process models. In: Bider I, Halpin T, Krogstie J, Nurcan S, Proper E, Schmidt R, Ukor R (eds) Enterprise, business-process and information systems modeling. Springer, Berlin, pp 157–168

Deutch D, Milo T (2007) Querying structural and behavioral properties of business processes. In: Arenas M, Schwartzbach MI (eds) 11th International symposium on database programming languages (DBPL 2007). Springer, Berlin, pp 169–185

Dijkman R, Dumas M, van Dongen B, Käärik R, Mendling J (2011a) Similarity of business process models: metrics and evaluation. Inf Syst 36:498–516

Dijkman R, Gfeller B, Küster J, Völzer H (2011b) Identifying refactoring opportunities in process model repositories. Inf Softw Technol 53:937–948

Dijkman R, La Rosa M, Reijers HA (2012) Managing large collections of business process models—current techniques and challenges. Comput Ind 63:91–97

Dumas M, García-Bañuelos L, Dijkman R (2009) Similarity search of business process models. IEEE Comput Soc Tech Comm Data Eng 32:23–28

Dumas M, García-Bañuelos L, La Rosa M, Uba R (2013) Fast detection of exact clones in business process model repositories. Inf Syst 38:619–633

Ekanayake C, Rosa M, Hofstede AM, Fauvet M-C (2011) Fragment-based version management for repositories of business process models. In: Meersman R, Dillon T, Herrero P, Kumar A, Reichert M, Qing L, Ooi B-C, Damiani E, Schmidt D, White J, Hauswirth M, Hitzler P, Mohania M (eds) On the move to meaningful internet systems: OTM 2011. Springer, Berlin, pp 20–37

Ekanayake CC, Dumas M, García-Bañuelos L, La Rosa M, ter Hofstede AHM (2012) Approximate clone detection in repositories of business process models. In: Barros A, Gal A, Kindler E (eds) Proceedings of the 10th international conference on business process management (BPM 2012). Springer, Berlin, pp 302–318

Fahland D, Jobstmann B, Koehler J, Lohmann N, Hagen V, Wolf K (2009) Instantaneous soundness checking of industrial business process models. In: Dayal U, Eder J, Koehler J, Reijers HA (eds) 7th International Conference on Business Process Management (BPM 2009). Springer, Berlin, pp 278–293

Fauvet MC, La Rosa M, Sadegh M, Alshareef A, Dijkman RM, García-Bañuelos L, Reijers HA, van der Aalst WMP, Dumas M, Mendling J (2010) Managing process model collections with AProMoRe. Service-Oriented Computing, pp 699–701

Feja S, Speck A, Witt S, Schulz M (2011) Checkable graphical business process representation. In: Catania B, Ivanović M, Thalheim B (eds) 14th East European conference on advances in databases and information systems (ADBIS 2010). Springer, Berlin, pp 176–189

Fettke P, Loos P (2005) Zur Identifikation von Strukturanalogien in Datenmodellen—Ein Verfahren und seine Anwendung am Beispiel des Y-CIM-Referenzmodells von Scheer. Wirtschaftsinformatik 47:89–100

García-Bañuelos L (2008) Pattern identification and classification in the translation from BPMN to BPEL. In: Meersman R, Tari Z (eds) Proceedings of the international conference on the move to meaningful internet systems (OTM'08). Springer, Berlin, pp 436–444

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman and Company, New York

Gottschalk F, Aalst WP, Jansen-Vullers M (2008a) Merging event-driven process chains. In: Meersman R, Tari Z (eds) On the move to meaningful internet systems (OTM 2008). Springer, Berlin, pp 418–426

Gottschalk F, van der Aalst WMP, Jansen-Vullers MH, La Rosa M (2008b) Configurable workflow models. Int J Coop Inf Syst 17:177–221

Hallerbach A, Bauer T, Reichert M (2009) Capturing variability in business process models: the provop approach. J Softw Maint Evol Res Pract 22:519–546

Hopcroft JE (1971) An n log n algorithm for minimizing states in a finite automaton. In: Kohavi Z (ed) Theory of machines and computations. Academic Press, London, pp 189–196

Hopcroft JE, Motwani R, Ullman JD (2008) Introduction to automata theory, languages, and computation. Pearson Education

Houy C, Fettke P, Loos P, Aalst WMP, Krogstie J (2011) Business process management in the large. Bus Inf Syst Eng 3:385–388

Huffman DA (1954) The synthesis of sequential switching circuits. J Frankl Inst 257:161–190

Jin T, Wang J, Wu N, La Rosa M, Ter Hofstede AHM (2010) Efficient and accurate retrieval of business process models through indexing. In: Meersman R, Dillon T, Herrero P (eds) Proceedings of the 2010 international conference on on the move to meaningful internet systems (OTM'10). Springer, Berlin, pp 402–409

Jin T, Wang J, Wen L (2011a) Efficient retrieval of similar business process models based on structure. In: Meersman R, Dillon T, Herrero P, Kumar A, Reichert M, Qing L, Ooi B-C, Damiani E, Schmidt D, White J, Hauswirth M, Hitzler P, Mohania M (eds) On the move to meaningful internet systems (OTM 2011). Springer, Berlin, pp 56–63

Jin T, Wang J, Wen L (2011b) Querying business process models based on semantics. In: Yu JX, Kim MH, Unland R (eds) 16th International conference on database systems for advanced applications (DASFAA 2011). Springer, Berlin, pp 164–178

Jin T, Wang J, Wen L (2012) Efficient retrieval of similar workflow models based on behavior. In: Sheng QZ, Wang GSC, Xu G (eds) 14th Asia-Pacific web conference (APWeb 2012). Springer, Berlin, pp 677–684

Knuplesch D, Ly L, Rinderle-Ma S, Pfeifer H, Dadam P (2010) On enabling data-aware compliance checking of business process models. In: Parsons J, Saeki M, Shoval P, Woo C, Wand Y (eds) 29th International conference on conceptual modeling (ER 2010). Springer, Berlin, pp 332–346

Koch I (2001) Enumerating all connected maximal common subgraphs in two graphs. Theor Comput Sci 250:1–30

Kunze M, Weske M (2011) Metric trees for efficient similarity search in large process model repositories. In: Muehlen M, Su J (eds) Business process management workshops. Springer, Berlin, pp 535–546

Kunze M, Weidlich M, Weske M (2011) Behavioral similarity—a proper metric. In: Rinderle-Ma S, Toumani F, Wolf K (eds) 9th International conference on business process management (BPM 2011). Springer, Berlin, pp 166–181

La Rosa M, Dumas M, Uba R, Dijkman R (2010) Merging business process models. In: Meersman R, Dillon T, Herrero P (eds) On the move to meaningful internet systems: OTM 2010. Springer, Berlin, pp 96–113

La Rosa M, Reijers HA, van der Aalst WMP, Dijkman RM, Mendling J, Dumas M, García-Bañuelos L (2011a) APROMORE: an advanced process model repository. Expert Syst Appl 38:7029–7040

La Rosa M, Dumas M, ter Hofstede AHM, Mendling J (2011b) Configurable multi-perspective business process models. Inf Syst 36:313–340

La Rosa M, Dumas M, Uba R, Dijkman R (2013) Business process model merging: an approach to business process consolidation. ACM Trans Softw Eng Methodol 22:1–42

Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. Sov Phys Dokl 10:707–710

Levi G (1973) A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo 9:341–352
Li C, Reichert M, Wombacher A (2009) Discovering reference models by mining process variants using a heuristic approach. In: Dayal U, Eder J, Koehler J, Reijers HA (eds) 7th International Conference on Business Process Management (BPM 2009). Springer, Berlin, pp 344–362
Li C, Reichert M, Wombacher A (2010) The minadept clustering approach for discovering reference process models out of process variants. Int J Coop Inf Syst 19:159–203
Lincoln M, Gal A (2011) Searching business process repositories using operational similarity. In: Meersman R, Dillon T, Herrero P, Kumar A, Reichert M, Qing L, Ooi B-C, Damiani E, Schmidt D, White J, Hauswirth M, Hitzler P, Mohania M (eds) International conference on the move to meaningful internet systems (OTM 2011). Springer, Berlin, pp 2–19
Lu R, Sadiq S, Governatori G (2009) On managing business processes variants. Data Knowl Eng 68:642–664
Mahleko B, Wombacher A (2006) Indexing business processes based on annotated finite state automata. In: 2006 IEEE International Conference on Web Services (ICWS'06). IEEE, Washington, pp 303–311
Marc W, Meinl T, Fischer I, Philippsen M (2005) A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In: Jorge AM, Torgo L, Brazdil P, Camacho R, Gama J (eds) 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005). Springer, Berlin, pp 392–403
McGregor JJ (1982) Backtrack search algorithms and the maximal common subgraph problem. Softw Pract Exp 12:23–34
Mendling J, Simon C (2006) Business process design by view integration. In: Eder J, Dustdar S (eds) Business process management workshops. Springer, Berlin, pp 55–64
Mendling J, Neumann G, van der Aalst WMP (2007) Understanding the occurrence of errors in process models based on metrics. In: Meersman R, Tari Z (eds) On the move to meaningful internet systems (OTM 2007). Springer, Berlin, pp 113–130
Mendling J, Verbeek HMW, van Dongen BF, van der Aalst WMP, Neumann G (2008) Detection and prediction of errors in EPCs of the SAP reference model. Data Knowl Eng 64:312–329
Momotko M, Subieta K (2004) Process query language: a way to make workflow processes more flexible. In: Benczúr A, Demetrovics J, Gottlob G (eds) 8th East European conference on advances in databases and information systems (ADBIS 2004). Springer, Berlin, pp 306–321
Nijssen S, Kok JN (2005) The gaston tool for frequent subgraph mining. Electron Notes Theor Comput Sci 127:77–87
Nijssen S, Kok JN (2006) Frequent subgraph miners: runtimes don't say everything. international workshop on mining and learning with graphs (MLG 2006), pp 173–180
Object Management Group (2003) Common warehouse metamodel 1.1. http://www.omg.org/spec/CWM/1.1/
Ouyang C, Dumas M, Arthur HM, van der Aalst WMP (2008) Pattern-Based translation of BPMN process models to BPEL web services. Int J Web Serv Res 5:42–62
Pascalau E, Awad A, Sakr S, Weske M (2011) On maintaining consistency of process model variants. In: Muehlen M, Su J (eds) Business process management workshops. Springer, Berlin, pp 289–300
Polyvyanyy A, Smirnov S, Weske M (2010) Business process model abstraction. In: Brocke J, Rosemann M (eds) Handbook on business process management 1. Springer, Berlin, pp 149–166
Qiao M, Akkiraju R, Rembert AJ (2011) Towards efficient business process clustering and retrieval: combining language modeling and structure matching. In: Rinderle-Ma S, Toumani F, Wolf K (eds) 9th International conference on business process management (BPM 2011). Springer, Berlin, pp 199–214
Rahm E, Bernstein PA (2001) A survey of approaches to automatic schema matching. VLDB J 10:334–350
Reijers HA, Mans RS, van der Toorn RA (2009) Improved model management with aggregated business process models. Data Knowl Eng 68:221–243
Reijers HA, Mendling J, Dijkman RM (2011) Human and automatic modularizations of process models to enhance their comprehension. Inf Syst 36:881–897
Rosemann M (2006) Potential pitfalls of process modeling: part A. Bus Process Manag J 12:249–254
Rosemann M, van der Aalst WMP (2007) A configurable reference modelling language. Inf Syst 32:1–23

Sakr S, Awad A (2010) A framework for querying graph-based business process models. In: Proceedings of the 19th international conference on World Wide Web (WWW'10). ACM Press, New York, pp 1297–1300

Scheer A-W (1992) Architecture of integrated information systems: foundations of enterprise modelling. Springer, Berlin

Scheidegger CE, Vo HT, Koop D, Freire J, Silva CT, Silva T (2008) Querying and re-using workflows with VsTrails. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data—SIGMOD'08. ACM Press, New York, pp 1251–1254

Shao Q, Sun P, Chen Y (2009) WISE: a workflow information search engine. In: 25th International conference on data engineering (ICDE'09). IEEE, Shanghai, pp 1491–1494

Sun S, Kumar A, Yen J (2006) Merging workflows: a new perspective on connecting business processes. Decis Support Syst 42:844–858

Thomas O, Fellmann MA (2009) Semantic process modeling—design and implementation of an ontology-based representation of business processes. Bus Inf Syst Eng 1:438–451

Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM 23:31–42

Van der Aalst WMP (1998) The APPLICATION OF PETRI NETS TO WORKflOW MANAGEMENT. J Circuits Syst Comput 08:21–66

Van der Aalst WMP (2013) Business process management: a comprehensive survey. ISRN Softw Eng 2013:1–37

Vardi MY (2007) Automata-theoretic model checking revisited. In: Cook B, Podelski A (eds) Verification, model checking, and abstract interpretation (VMCAI 2007). Springer, Berlin, pp 137–150

Wang J, Jin T, Wong RK, Wen L (2013) Querying business process model repositories—a survey of current approaches and issues. World Wide Web

Weber B, Reichert M, Mendling J, Reijers HA (2011) Refactoring large process model repositories. Comput Ind 62:467–486

Weidlich M, Mendling J, Weske M (2011) A foundational approach for managing process variability. In: Mouratidis H, Rolland C (eds) 23rd International conference on advanced information systems engineering (CAiSE 2011). Springer, Berlin, pp 267–282

Welling R (2011) A performance analysis on maximal common subgraph algorithms. In: 15th Twente Student Conference on IT., Enschede, Netherlands

Wynn M, Verbeek HMW, van der Aalst WMP, ter Hofstede AHM, Edmond D (2009) Business process verification—finally a reality! Bus Process Manag J 15:74–92

Yan X, Han J (2002) gSpan: graph-based substructure pattern mining. IEEE international conference on data mining. IEEE Comput. Soc, Maebashi City, Japan, pp 721–724

Yan Z, Dijkman R, Grefen P (2012) Fast business process similarity search. Distrib Parallel Databases 30:105–144