Towards Reuse of Business Processes Patterns to Design Services

Veronica Gacitua-Decar and Claus Pahl

Abstract. Service Oriented Architecture is a promising architectural approach to solve the integration problem originated by business process integration and automation requirements. Defining the appropriate granularity and scope of services is a critical issue to allow their reuse. Architecture abstractions, such as patterns, are a medium to capture design knowledge and to allow the reuse of successful previous designs. The continual rise of abstraction in software engineering approaches have been a central driver of this work, placing the notion of patterns at business model level. In this paper we propose a set of pattern-based techniques to define the scope and granularity of services based on identified patterns in business process models. Graph-based pattern matching and pattern discovery are proposed to recommend the scope and granularity of services on process-centric description models. Matching of generalised patterns and hierarchical matching are discussed.

Keywords. service oriented architecture, business process pattern, service design, service identification, pattern matching, pattern discovery.

1. Introduction

Nowadays, evermore organizations are taking advantage of consolidating relations with service provider companies in order to improve competitiveness. This involves the merging of internal processes from provided and provider companies into inter-organisational processes shaped by a business chain value [3]. At technical level, business process integration creates an Enterprise Application Integration (EAI) problem. Service-Oriented Architecture (SOA) has appeared as a promising architectural approach to solve the EAI problem generated during processes integration and automation. Defining the scope and granularity of services is a critical issue to benefit from the advantages of implementing a SOA approach, in particular service *reuse* [9]. Defining the scope of services involves the analysis of business models and the existing software support. Existing software support might already be implemented as services, but most frequently it still is provided as

legacy applications. Thus, the *identification* of services involved in the architecture solution might consider the *discovery* of existing services, but most frequently, the *definition of new services*.

Reuse of services within the limits of one organisation and its partners, providers and clients in close cooperation can be exploited by planning in advance the services that will be available. In this manner, *reuse* of services is emphasised at design time - before implementation. This is specially relevant for large organisations where overlapping functionality offered by different services can rapidly grow, overshadowing the benefits of service reuse.

Architecture abstractions like patterns and styles can capture design knowledge and allow the reuse of successfully applied designs and improve the quality of software [7]. Abstraction in software engineering approaches is a central driver; at the business level the reuse of successfully business designs is equally important. However, the abstraction and reuse principles associated to patterns have not been exploited enough to design new services based on patterns defined at business model level.

A number of contributions have addressed the problem of service identification. High level guidelines to design new services such as in [9] are very useful, however they require advances regarding formality and techniques that can be finally materialised as tool support. There are approaches, such as [21] and [8], that have proposed techniques to automate the discovery of services by matching process-centric descriptions - beyond the matching of service signatures and effects. They base their solutions on the comparison of a requested process-centric description against descriptions of existing services in a service repository. However, when defining new process-centric services, their boundaries are defined over sections of business process models. These processes are often larger and more complex than the description of single services. Additionally, several contributions have investigated solutions to compare or to query process-centric model descriptions [2], [4], [6]. They differ in focus, expected results and performance issues, and no one of them has investigated the idea of defining the scope and granularity of process-centric services based on identifying the occurrence of business process patterns in business process models.

In this paper we present a set of pattern-based techniques and algorithms focused on the identification of business process pattern instances in process-centric models. These instances are used to recommend the scope and granularity of new process-centric services. Note that the discovery of existing services has not been directly addressed here, however the proposed algorithms and related concepts could contribute to graph-based techniques used to match service descriptions, such as for example the work in [10], and to complement and to encourage the reuse of process patterns as is reinforced in [15].

- The definition of *new process-centric services* is addressed by means of a hybrid approach combining *structural matching* of business process patterns in business process models, and the use of a *controlled vocabulary* - specific to business domains. The later relaxes a pure syntactic matching for labels associated to process elements. The latter is discussed in the paper as matching of *generalised patterns*. *Hierarchical* matching allows incremental levels of abstraction for matched patterns. *Partial*

pattern matching provides flexibility to the proposed techniques. *Inexact* pattern matching is discussed.

- The other technique presented here exploits the fundamental principle of *reuse* in the scenario where a pattern repository does not exist and patterns can not be matched, but rather, they need to be discovered. The intuitive idea is to find *frequent* process substructures -named *utility patterns*- within large process models. Frequent set of organised process steps might be supported by existing software components, which can be rationalised, and subsequently encapsulated as reusable technical-centric services.

The remainder of this paper is organised as follows. Section 2 introduces a graph-based representation for process models, process patterns and their relation during *pattern instantiation*. Section 3 describes the different aspects of the process pattern matching problem and our proposed solutions. Section 4 describes our proposal for finding utility patterns in process models. Section 5 provides an evaluation of the proposed exact and partial pattern matching techniques. Finally, in sections 6 and 7 a review of related work and conclusions is provided.

2. Graph-based representation of Business Process Models and Business Process Patterns

Graphs emerge as a natural representation for process-centric models [11],[18]. Graphs can capture both structure and behaviour, and allow abstractions such as patterns to be related to process-centric models.

2.1. Structural Representation of Business Process Models as Graphs

In the context of this paper we use graphs to represent the structure of process models and process patterns. Graph vertices represent process elements such as activities and control flow elements. Graph edges represent the connectivity between process elements. Labels and types of graph vertices represent names and types of process model elements. Section 8 (annex) provides an introductory background on graphs and the related notation used in this section and along the rest of the paper.

Graph-based business process model.
Let the graph $PM = (V_{PM}, E_{PM}, \ell_{V_{PM}}, \ell_{E_{PM}})$ be a finite, connected, directed, labelled graph representing a business process model. V_{PM} is the set of vertices representing process elements and E_{PM} is the set of edges representing *connectivity* between process elements. The function $\ell_{V_{PM}} : V_{PM} \rightarrow L_{V_{PM}}$ is the function providing labels to vertices of PM, and $\ell_{E_{PM}} : E_{PM} \rightarrow L_{E_{PM}}$ is the function providing labels to edges of PM. $L_{V_{PM}}$ and $L_{E_{PM}}$ are the sets of labels for vertices and edges, respectively. Note that in this paper, *connectivity* between process elements is simplified by considering only the sequence flows between activities since we emphasise the matching of *structural* relations between patterns and process. The addition of input, output, pre and post condition information associated to process steps could be captured on edges attributes, but this is not addressed in this here.

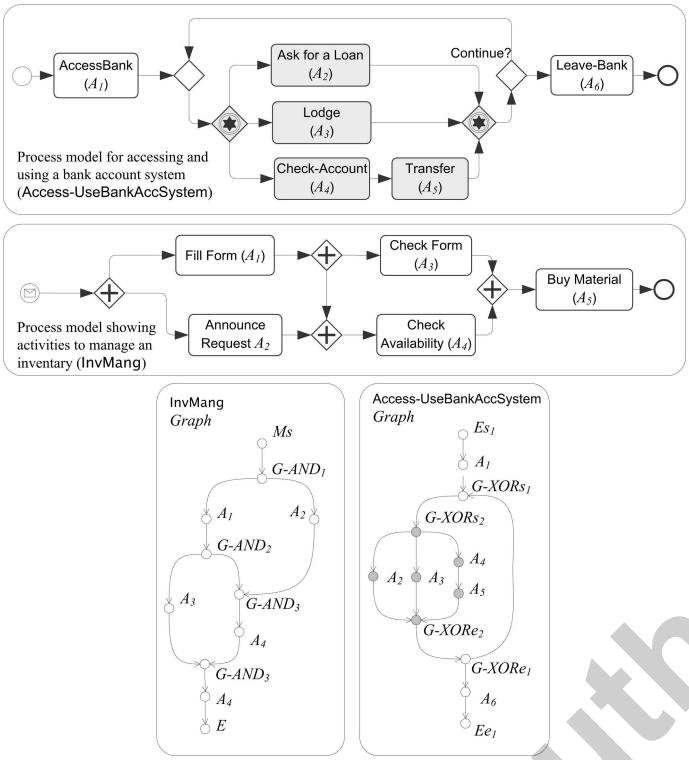

FIGURE 1. Two example of process models annotated with BPMN and related graph-based representations.

Fig. 1 provides an example of an intuitive graph-based representation of business process models annotated with a well-known process modelling notation, i.e. Business Process Modelling Notation[1] (BPMN). An appropriate mapping function maps descriptions of process elements to graph labels. Note that similar graph-based models can represent executable processes described, for instance, in the standard WS-BPEL language[2], however the latter requires additional considerations due to the block-based structure of the language [20]. An example of a graph-based representation for a WS-BPEL process is illustrated in Fig. 2.

2.2. Structural Representation of Business Process Patterns as Graphs

Business Process (BP) patterns are essentially common connectivity patterns in process models. BP patterns can be operator-oriented, e.g. a multi-choice pattern that allows the selection of a number of options instead of an exclusive selection based on the basic choice operator. This kind of process patterns are know in the literature as *workflow patterns* [1]. Other category of BP patterns consists of application context-oriented and often

[1] Available from http://www.bpmn.org/Documents/BPMN 1-1 Specification.pdf
[2] Available from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

```
<bpel:process name="example" …">
<bpel:import … />
…
 +<bpel:partnerLinks>…</bpel:partnerLinks>
 +<bpel:variables>…</bpel:variables>
 -<bpel:scope>
  -<bpel:faultHandlers>
   -<bpel:catch faultMessageType="ns1:Exception" …>
    +<bpel:sequence>
     +<bpel:assign>…</bpel:assign>
      <bpel:reply name="Reply_S2" operation="…" …/>
    </bpel:sequence>
   </bpel:catch>
  </bpel:faultHandlers>
  -<bpel:flow>
   -<bpel:links>
     <bpel:link name="L1"/>
     <bpel:link name="L3"/>
     <bpel:link name="L2"/>
     <bpel:link name="L4"/>
   </bpel:links>
   -<bpel:receive … name="Receive_S1" …>
    -<bpel:sources>
      <bpel:source linkName="L1"/>
      <bpel:source linkName="L3"/>
    </bpel:sources>
   </bpel:receive>
   -<bpel:reply name="Reply_S3" …>
    -<bpel:targets>
      <bpel:target linkName="L2"/>
      <bpel:target linkName="L4"/>
    </bpel:targets>
   </bpel:reply>
   -<bpel:while>
    -<bpel:targets>
      <bpel:target linkName="L1"/>
    </bpel:targets>
    -<bpel:sources>
      <bpel:source linkName="L2"/>
    </bpel:sources>
    <bpel:condition>…</bpel:condition>
    +<bpel:sequence>
     -<bpel:assign>…</bpel:assign>
      <bpel:invoke … name="Invoke_I1" …/>
     -<bpel:assign>…</bpel:assign>
    </bpel:sequence>
   </bpel:while>
   -<bpel:sequence>
    -<bpel:targets>
      <bpel:target linkName="L3"/>
    </bpel:targets>
    -<bpel:sources>
      <bpel:source linkName="L4"/>
    </bpel:sources>
    -<bpel:assign>…</bpel:assign>
      <bpel:invoke … name="Invoke_I2" …/>
    -<bpel:assign>…</bpel:assign>
   </bpel:sequence>
  </bpel:flow>
 </bpel:scope>
</bpel:process>
```

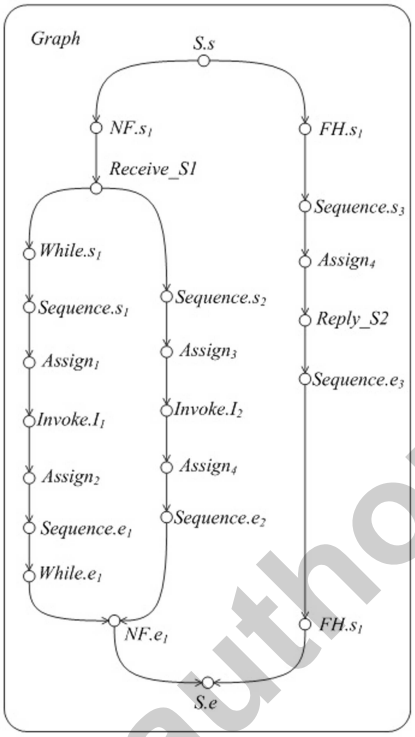

FIGURE 2. Excerpt of executable WS-BPEL process and a related graph-based representation.

more complex patterns derived from and specific to the business context. This kind of BP patterns can represent well-known process building blocks in reference models, abstracting a set of connected activities required to reach some business goal [5],[12]. Application context-oriented business process patterns can be reused as previously implemented and successful designs and provide an integrated vision of processes among different participants. For instance, in the Fig. 1 the Use-AccessBankAccSystem process has at its core, in gray colored vertices, a common set of account usage activities that can be represented in the form of a application-context oriented process pattern.

Beyond the previous types of BP patterns, a third category represent frequent process connectivity structures that are not specific to a business domain. They often relate to some standard technology solution, for instance an authentication and authorisation process to access a system. We name these pattern *utility* patterns, borrowing the name from

the definition of *utility* services in [9]. In the rest of the paper we will refer to *application context-oriented business process patterns* only as *patterns*. *Workflow patterns* are not addressed here. *Utility patterns* are the focus of Section 4.

Graph-based business process pattern.
Let the graph $PP = (V_{PP}, E_{PP}, \ell_{V_{PP}}, \ell_{E_{PP}})$ be the finite, connected, directed, labelled graph representing a business process pattern model. Elements of V_{PP} represent process pattern roles and elements in E_{PP} represent connectivity between pattern roles. Note that the graph-based representation for business patterns, utility patterns and business processes is structurally the same.

2.3. Instantiation of Process Patterns in Process Models

Process patterns have been described in the same way as process models. Now, we discuss the relation between process patterns and process models. In particular, we are interested in the abstraction that patterns represent for process models and concretely, in the notion of *instantiation* of process patterns in process models.

Pattern Instantiation in a concrete model indicates that the structural relations described in the pattern hold in the model. The structural preserving relations that graph homomorphisms represent help us to capture the notion of pattern instantiation. In particular, instantiation of a BP pattern in a process model can be captured by the definition of a *locally surjective graph homomorphism* [13] between a subgraph PM_S of the graph process model PM and the pattern graph PP, i.e. $PM_S \xrightarrow{S} PP$. *Surjection* allows that several process elements (vertices of PM) can play the role of one pattern element (vertex of PP) - see illustration[3] in the Fig. 3. Moreover, model elements can belong to more than one pattern when considering this approach.

3. Process Pattern Matching

We have discussed in Section 1 the potential that discovering instances of patterns in concrete models can provide to the definition of new services. Matching a pattern in a concrete model involves the identification of instances of that pattern in the concrete model. In this manner, as was previously explained in general terms, the *process pattern matching problem* can be referred as the detection of a graph homomorphism between the graph representing a concrete model and the graph representing the pattern.

3.1. Exact, Inexact and Partial Pattern Matching

In realistic scenarios where an *exact* match of a pattern is unlikely, *partial* and *inexact* matching become relevant. *Inexact pattern matching* provides *good*, but not *exact* solutions to the matching problem. In this case, pattern instances can incorporate additional elements not described in the pattern, nevertheless they must not affect the structural properties of the pattern. *Partial pattern matches* identify exact but *incomplete* matches of patterns. Partial instances of patterns might exist due to modifications or evolution of

[3] Note that we have used the notation from the previous section and the Annex.

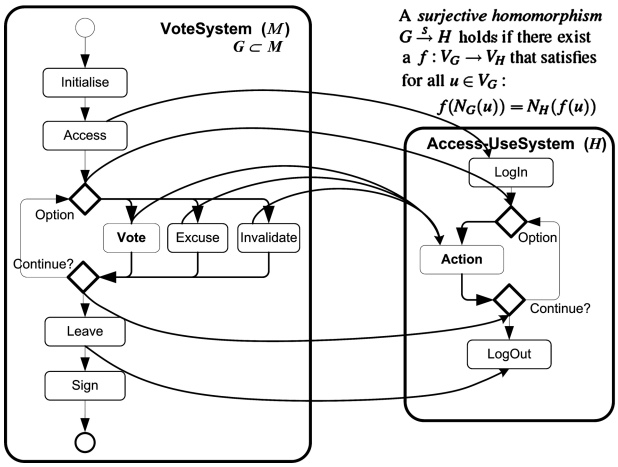

FIGURE 3. Illustration of several instances (Vote, Excuse, Invalidate) of a pattern role (Action) in a concrete process model.

previously instantiated patterns. However, when patterns have not previously considered as part of the design, partial matches might indicate an opportunity to improve the design by means of incorporating the whole pattern. *Partial* and *inexact* matches are also important due to the fact that process models and their implementations as services might be highly similar but not exactly the same from organisation to organisation and to identify commonalities can save costs and encourage reuse. Fig. 4 illustrates examples of *exact*, *partial* and *inexact* pattern instances that can potentially be matched.

In order to formalise and later on to implement our proposed techniques as concrete tool support, we will define exact, partial and inexact pattern matching in terms of the graphs representing processes and patterns and their structural relations. Formalisation can provide guaranties of correctness and improve the confidence in tools.

Exact Pattern Matching. An exact pattern match of a specific pattern PP in an arbitrary process model PM refers to the detection of a surjective subgraph homomorphism, i.e. $PM_S \xrightarrow{S} PP$, where $PM_S \subseteq PM$. The mapping function φ defines an individual instantiation of the pattern PP in the process model PM, where $\varphi : V_{PM_S} \rightarrow V_{PP}$ satisfying that for all $u \in V_{PM_S} : \varphi(N_{PM_S}(u)) = N_{PP}(\varphi(u))$, and with the mapping $\lambda_S : L_{V_{PM_S}} \rightarrow L_{V_{PP}}$ a bijective function indicating a semantic correspondence between the labels of two mapped vertices. Note that in Fig. 4, the example of an exact pattern instance (A) not only illustrates a semantic correspondence between the labels in the pattern and the process model, but also a syntactical one. The example showing an inexact pattern instance in Fig. 4(C) considers a case where semantic correspondence holds, but not a syntactical one, i.e. the case between the Lodge and Deposit activities.

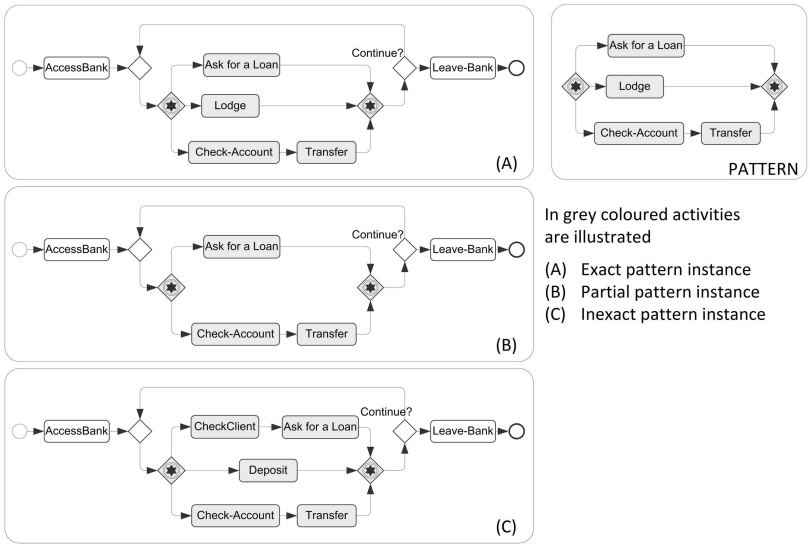

FIGURE 4. Exact, partial and inexact pattern instances.

Partial pattern matching. Partial matches restrict the matching problem allowing incomplete matches. Incomplete pattern matches maps elements from PM to a reduced number of elements considered in the original codomain (V_{PP}). In this manner, the original function φ defined for exact matching is now restricted to the function $\varphi_{PARTIAL} : V_{PM_{S^*}} \rightarrow V_{PP_{PARTIAL}}$ satisfying that for all $u \in V_{PM_{S^*}}$ $\varphi_{PARTIAL}(N_{PM_{S^*}}(u)) = N_{PP_{PARTIAL}}(\varphi_{PARTIAL}(u))$ where $PP_{PARTIAL} \subseteq PP$ and $PM_{S^*} \subseteq PM_S$.

Inexact pattern matching. Inexact pattern matching relaxes the definition of *neighborhood* for vertices in a graph. The Annex (Section 8) provide more details. The set of *neighbors* vertices to a vertex u is now considered as $N^*_{PM_S}(u)$ allowing other vertices not only in the original neighborhood of u ($N_{PM_S}(u)$) but also in the *path* between u and v with $\varphi(u)$ adjacent with $\varphi(v)$ and $\varphi : V_{PM_S} \rightarrow V_{PP}$. Borrowing the name from [19], we call these vertices - the vertices that are not in the original neighborhood - *intermediate* vertices. An example of an intermediate vertex is illustrated with the CheckClient activity in Fig. 4(C).

Algorithm for Exact and Partial Matching. We propose an algorithm for exact and partial process pattern matching. The pseudo-code of the proposed algorithm is described in Table 1 (ALGORITHM 1 - *u*EP-PMA). The algorithm starts matching each vertex in V_{PP} with vertices in V_{PM} such that the labels in $L_{V_{PP}}$ semantically correspond to labels in $L_{V_{PM}}$. Semantic correspondence in *u*EP-PMA refers to a one to one (bijective) mapping λ. The mapping λ is a mapping between a subset of labels in $L_{V_{PM}}$ and labels in $L_{V_{PP}}$. The subset in $L_{V_{PM}}$ correspond to labels of matched vertices in V_{PM}. Each initial match is considered a temporal pattern match defining a (temporal) subgraph in PM that we denote as tPM.

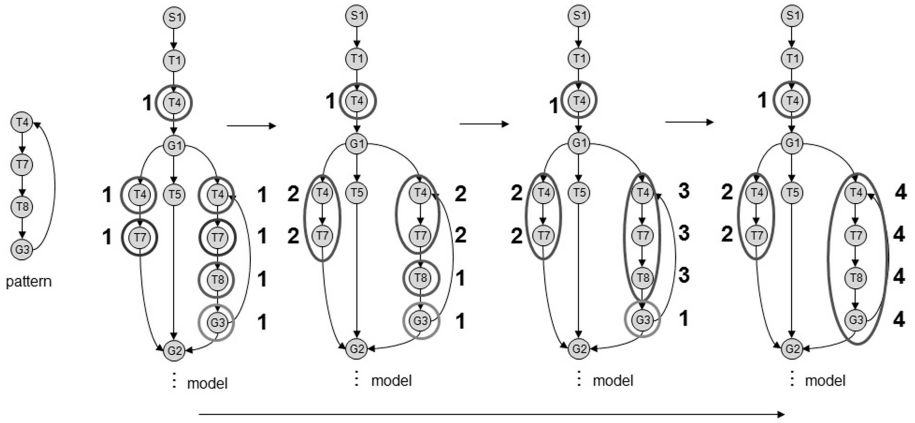

FIGURE 5. Matching expansion steps. One exact match and two partial matches are found.

Subsequently, tPM is expanded until all its neighbors that hold a structural relation defined by φ or at least $\varphi_{PARTIAL}$ are added. Fig. 5 illustrates the expansion steps from the initial set of tPM, in this case, vertices labelled as $T4, T7, T8$ and $G3$. The algorithm terminates when no more expansion steps can be done. The result is a *score* vector. Each vertex in PM has a score that indicates the number of vertices of the matched pattern to which it belongs.

Note that several exact or partial instances of PP in PM might exist. If different pattern instances share edges in PM, we say that there are *overlaps* of the pattern PP in PM. The *u*EP-PMA algorithm identifies the connected subgraphs in PM containing overlaps as a one single subgraph PM_O. The score of the vertices in the overlap is the number of vertices in PM_O. Additionally, in order to consider the directionality of the graphs representing concrete models and patterns, the *u*EP-PMA algorithm can also be performed on the *undirected* version of PM and PP, which is indicated with a *u* in **ALGORITHM 1** (Table 1). In this manner, matches not only considers vertices, but also arcs.

According to [14], for a connected simple graph H, the problem of detecting a locally surjective homomorphism between an arbitrary graph and H is solvable in polynomial time if and only if H has at most two vertices. In all other cases the problem is NP-complete. The complexity of the latter problem, which is directly related to the pattern matching problem, made us aware of performance issues. In Section 5 we show a preliminary evaluation where instances of specific graph patterns are identified on arbitrary random graphs. The results show that the time required to solve the problem is quadratic in relation to the size of the random graphs and it has a small constant that conveniently modulates the response time for small and medium size graphs. Scalability, in terms of processing several patterns over one or more target graphs, could be addressed by implementing a refined version of the algorithms to allow parallel processing for each pattern.

ALGORITHM 1: *u*EP-PMA.

Input: Target Graph (*PM*), Pattern Graph (*PP*)
Output: Score Vector (*score*).

For each vertex m in V_{PM} **do**
  For each vertex p in V_{PP} **do**
    If $\lambda \circ \ell_{V_{PM}}(m) = \ell_{V_{PP}}(p)$ == true **then**
      $tPM(m) \leftarrow$ initial temporal match centred in vertex $m \in PM$
      $score \leftarrow 1$ (*score* for vertices in $tPM(m)$)
    end if
  end for
end for
Do while *ExpansionCondition* == true
  For each vertex $i \in tPM(m)$ **do**
    If $\ell^{-1}_{V_{PP}} \circ \lambda \circ \ell_{V_{PM}}(N_{tPM(m)}(i)) = N_{PP}(\ell^{-1}_{V_{PP}} \circ \lambda \circ \ell_{V_{PM}}(i))$ && $N_{tPM(m)}(i) \notin tPM(m)$ **then**
      Expand $tPM(m)$ with $N_{tPM(m)}(i)$
      $score \leftarrow score + 1$
      ExpansionCondition $\leftarrow$ true
      Else if
        ExpansionCondition $\leftarrow$ false
      end else if
    end if
  end for
end do while

TABLE 1. *u*EP-PMA - (*u*ndirected) Exact and Partial - Pattern Matching Algorithm.

3.2. Matching of Generalised Patterns

Considering a restricted vocabulary for different vertical business domains can add additional benefits to the practical use of BP pattern matching solutions. There are cases where descriptions of process elements (or pattern elements) have the same syntax, but different semantics and vice versa. Moreover, processes and patterns might be described with different structures, while they behave in the same way [2]. Regarding the vocabulary used to describe process and pattern elements, we have extended the *u*EP-PMA algorithm with the *u*G-PMA algorithm allowing semantic correspondence beyond the one to one mapping (λ) previously considered. The structure of the algorithm remains relatively invariant, but the functions $\ell_{V_{PM}}$, $\ell_{V_{PP}}$ and λ are modified. Labels in the taxonomy refer to concepts from a particular business domain. The extension modifies the two $\ell_{(\cdot)}$ functions in order to map vertices from V_{PM} to labels that are organised in a tree-like structured taxonomy. Labels in $L_{V_{PM}}$ can be mapped to specialisations of or equivalent labels in $L_{V_{PP}}$. In this manner, *generalised patterns* are considered as families of patterns where the parent pattern contains the roots of tree-structured taxonomies for business concepts in specific domains. Child patterns contains one or more child concepts connected to root concepts

in the hierarchy defined by the taxonomy. Note that using the *u*G-PMA algorithm requires the existence of an implemented taxonomy in which the algorithm can search for semantically corresponding terms.

3.3. Hierarchical Pattern Matching

In the previous sections we have addressed the exact and partial matching problem on flat process models (and patterns). However, processes and patterns are commonly composed by more fine-grained process-centric structures. In this section we outline a solution to the problem of pattern matching considering different levels of abstraction.

Algorithm for hierarchical pattern matching. The pseudo code of the proposed algorithm named *u*H-PMA is described in **ALGORITHM 2**. The algorithm starts matching different patterns (PP_j) from a set of patterns (*setPP*) at a certain level of granularity on a target model *PM*. Note that the index j identifies an specific pattern in *setPP*. After perform the initial matches, *PM* is transformed to an abstracted representation PM^i, where i represents a particular level of abstraction. In a particular PM^i, subgraphs in the previous level of abstraction (PM^{i-1}) that are associated to matches of a PP_j are replaced by vertices $\mathtt{p}_j$ whose type is '*pattern*'[4]. Thus, the complexity of a matched subgraph is hidden in a vertex $\mathtt{p}_j$ whose type is *pattern*. Note that representative labels are assigned to pattern vertices. Once the target model is abstracted by replacing matches with *pattern* vertices at a specific level of abstraction, new patterns at a higher level might appear. In this way, the abstraction process can be performed iteratively, abstracting a process graph PM^i into a process graph PM^{i+1} which is one level of abstraction up, and so on. The algorithm terminates when no more matches are found or when the process graph has only one vertex. Note that this process requires graphs excluding overlaps.

Fig. 6 illustrates the idea of hierarchical pattern matching. It uses the process model from Fig. 1 and shows two patterns, BankAccUsage and Access-UseSystem which are consecutively matched at two different levels of abstraction. The pattern BankAccUsage describes a set of common bank account usage activities and the pattern Access-Use System represents a typical -simplified- set of steps to access a generic system. Note that BankAccUsage is focused on the banking industry, however the Access-UseSystem pattern can be valid across different industries since it has a technology-oriented and business-agnostic nature [12]. The result of the hierarchical pattern matching process is a single vertex representing the access and use of a system.

Note that we have not addressed the problem of *overlaps* yet, i.e. how to abstract two matches that share vertices and edges in the target model? Our basic representation of processes and patterns as graphs restricts the possibility of representing two overlapped matched patterns as two different pattern vertices. One idea that we are exploring is the representation of matched patterns as hyperedges of a hypergraph. The vertices of the hypergraph are the same vertices of the graph representing the process model.

[4]Typed graphs hold a complete mapping to a set of types. Mappings for typed graphs can consider vertices and edges. The mapping function for $\mathtt{p}_j$ is a global surjective function from the set of graph vertices to a set of types that classify patterns, and whose parent type is '*pattern*'.

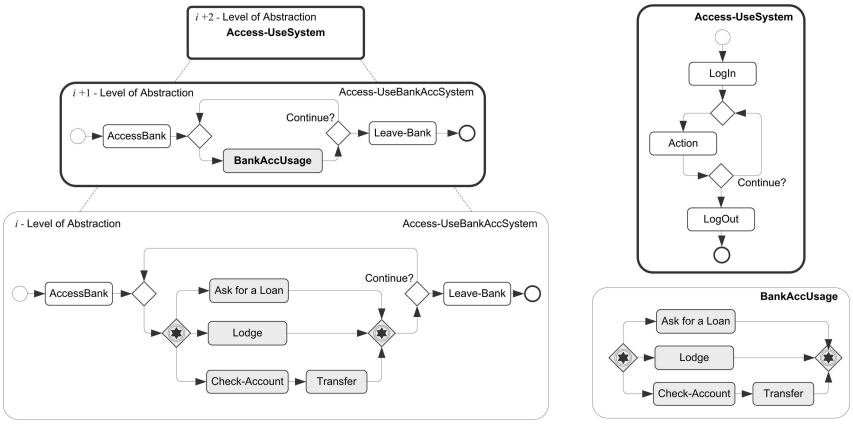

FIGURE 6. Hierarchical pattern matching.

ALGORITHM 2: *u*H-PMA.

Input: Target Graph (PM), Set $setPP$ of n pattern graphs ($setPP = \{PP_1, ..., PP_n\}$)
Output: $scoreMatrix$[5].

1 : **Do while** $IterationCondition$&&$change$ == true
2 : **For each** pattern $PP_j \in setPP$ **do**
3 : *u*EP-PMA(PM^i, PP_j) (or *u*G-PMA if generalised pattern matching is desired)
4 : **If** $score(u) = |V_{PP_j}|$ with $u \in PM^i_{S_j}$ && $exact\ match == true$ **then**
5 : $PM^i_{S_j} \leftarrow \mathrm{p}_j$
6 : $change \leftarrow$ true
7 : **If** $|V_{PM^i}| <= 1$ **then**
8 : $IterationCondition \leftarrow$ false
9 : **end if**
10 : **end if**
11 : **Else if**
12 : $change \leftarrow$ false
13 : $i \leftarrow i+1$
14 : **end for**
15 : **end do while**

TABLE 2. *u*EP-PMA - (*u*ndirected) Hierarchical - Pattern Matching Algorithm.

4. Discovering Frequent Utility Patterns in Process Models

Previous sections described techniques for identifying services based on the matching of known application context-oriented process patterns in process models. In this section we are interested in discovering frequently occurring substructures on large scale business process models. Process steps might be supported by existing software components

and identifying reoccurring connected process steps provide a medium to define potential reusable software components as encapsulated services. The idea is to exploit the basic principle of reuse in SOA. Finding frequent -not necessarily known- *utility patterns* in large process models can help to the definition of reusable technical-centric services.

There are two distinct problem formulations for frequent pattern discovery in graphs: graph-transaction setting and single-graph setting [17]. The latter refers to the discovery of subgraphs that occur multiple times in a single input graph. The other refers to the discovery of subgraphs that occur frequently across a set of small graphs. We present an algorithm focused on the single-graph setting scenario for pattern discovery in graphs.

Algorithm for Pattern Discovery. The aim of the proposed algorithm is to find frequent -exact and partial- occurrences of subgraphs in a single input graph PM. A discovered frequent subgraph -utility pattern- is an induced subgraph PP_U homomorphic to all occurrences of a frequent subgraph in PM. Homomorphism detection in the proposed algorithm (named uEP-FPDA) relies on the pattern matching algorithm (uEP-PMA) described in Section 3.1. The pseudo code of uEP-FPDA is described in Table 3.

The size of the induced subgraphs and a parameter that relaxes the way of counting the frequency of the induced subgraphs are parameterised by k and Th, respectively. The constant k refers to the amount of times that an initial subgraph in PM will be *expanded* and compared to other subgraphs in PM in order to check for homomorphisms. Th refers to a threshold for the ratio between the number of vertices of two non exact occurrences of PP_U. If Th is equal to one, the occurrences must be isomorphic between them.

The output of uEP-FPDA are two matrices, *score* and $FreqM$. Rows in the *score* matrix represent each vertex u in PM. Columns represent the results of the algorithm for different pattern sizes[6]. If u belongs to a highly frequent subgraph in PM of size j then $score(u, j)$ will be also high. $FreqM$ is a matrix with $|V_{PM}|$ rows and k columns, where each cell indicates the frequency of a discovered pattern. The row index indices where the pattern is centred, i.e. a vertex in V_{PM}. The column index (k) indicates the size of the discovered pattern.

The uEP-FPDA algorithm starts defining an arbitrary vertex u from the target graph PM as the first temporal pattern (pivot pattern or $PP_{pivot(u,1)}$) and then it matches $PP_{pivot(u,1)}$ against the rest of the target graph. The matrices *score* and $FreqM$ are initialised and the results of the first matches (for patterns with size equal to 1) are annotated. The next steps are repeated for each vertex in PM. The first matched vertices are called $seeds_{(u,1)}$. The subgraph $PP_{pivot(u,1)}$ and each of the previously matched vertices are expanded with their neighbors. The algorithm continues the expansion of PP_{pivot} by checking if a homomorphism between the expanded PP_{pivot} and subgraphs in PM holds. The expansion process continues for k times -external parameter- or until no more homomorphisms are detected. The results contained in *score* and $FreqM$ indicate the set of induced subgraphs PP_U and they represent the discovered utility patterns.

[6] Size of a pattern is considered as the number of vertices in that pattern

ALGORITHM 3: *u*EP-FPDA.

Input: Target Graph - undirected version (uPM), Threshold (Th), number of expansion steps (k)
Output: $score$, $FreqM$

```
1 : For each vertex u in uPM do
2 :   PP_pivot(u,1) ← u
3 :   seeds_(u,1) ←uEP-PMA(PM, PP_pivot(u,1))
4 :   score_(u,1) ← seeds_(u,1)
5 :   For each i in seeds_(u,1) do
6 :     If score_(u,1)(i)/|PP_pivot(u,1)| >= Th then
7 :       cnt_(u,1) ← cnt_(u,1) + 1
8 :     end if
9 :   end for
10 :  FreqM(u,1) ← cnt_(u,1)/|PP_pivot(u,1)|
11 :  If k >= 1 do
12 :    For j : 2 → k
13 :      PP_pivot(u,j) ← expand(PP_pivot(u,j−1))
14 :      seeds_(u,j) ←uEP-PMA(PM, PP_pivot(u,j))
15 :      score_(u,j) ← seeds_(u,j)
16 :      For each i in seeds_(u,j) do
17 :        If score_(1)(u, j)/|PP_pivot(u,j)| >= Th then
18 :          cnt_(u,j) ← cnt_(u,j) + 1
19 :        end if
20 :      end for
21 :      FreqM(u, j) ← cnt_(u,j)/|PP_pivot(u,j)|
22 :    end for
23 :  end if
24 : end for
```

TABLE 3. *u*EP-FPDA - (*u*ndirected) Exact and Partial - Frequent Pattern Discovery Algorithm.

Based on the results obtained in a preliminary evaluation (see Section 5), where the *u*EP-PMA algorithm exhibits a complexity of quadratic order, it is expected that for *u*EP-FPDA the complexity will grow up to $O(kV^3)$, where V is the the number of vertices in the target graph (undirected version), and k is the number of times the temporal patterns in *u*EP-FPDA are expanded.

5. Evaluation

We have performed a preliminary evaluation for the exact and partial matching algorithm (*u*EP-PMA). The experiments consider seven specific patterns over arbitrary random graphs with approximate sizes of 60, 450, 1300, 1800, 3200 and 5000 vertices. The experiments were run on a Intel machine 2 GHz and 2GB RAM on WinXP-SP3. In patterns and random graphs three different types of labels were considered, A, B or C. The

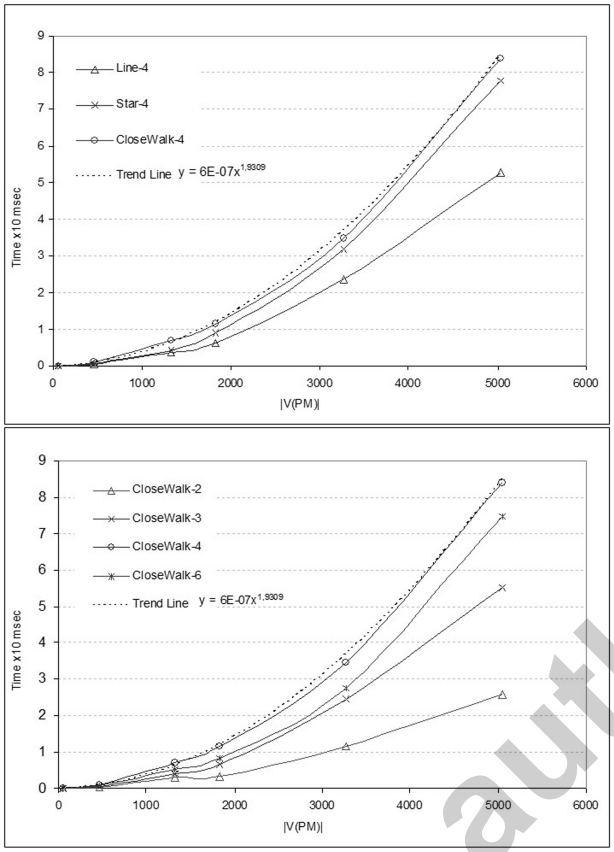

FIGURE 7. Average response time of *u*EP-PMA on arbitrary random graphs for different pattern structures (left side) and different pattern sizes (right side).

considered patterns encompassed four close-walks of 2, 3, 4 and 6 vertices; two line-like patterns of 3 and 4 vertices and a star-like pattern of 4 vertices.

Fig. 7 (top) shows the average response time of *u*EP-PMA when matching three patterns with different structures and the same number of vertices. The line-like pattern requires less time in comparison to the star-like and close-walk patterns. It indicates that the structure of matched patterns influences the response time. Fig. 7 (bottom) shows the average response time of *u*EP-PMA for patterns with the same structure (close-walk in this case) and different number of vertices. The number of vertices in the pattern also influence the time response. In order to visualise the time response trend more clearly, we divided the time required by the algorithm to compute a solution by the ratio between the number of vertices in the random graph (target graph) and the number of vertices in the pattern. Fig. 8 (left side) illustrates the trend of the normalised response time for

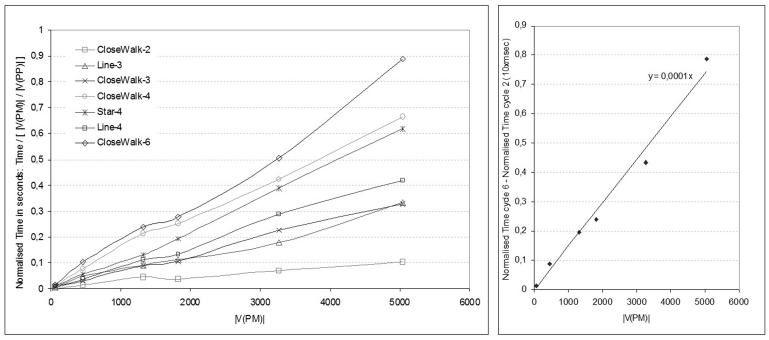

FIGURE 8. Average response time of *u*EP-PMA algorithm on arbitrary random graphs for matching a star-like pattern, a line-like pattern and a pattern with a close-walk structure.

all different patterns considered in the experiment. The right side of Fig. 8 illustrates the trend of the normalised time response for two patterns with different number of vertices. The trend lines in Fig. 7 indicate that the time to solve the problem increases quadratically with the number of vertices in the target graph. The constant 6^{-7} suggest advantageous performance characteristics regarding the response time of the algorithm for small and medium size graphs. Note that the performed experiments are preliminary, in the sense that they consider simple labels for graphs vertices. Moreover, highly structured graphs such as the case for process models have not been considered yet and only a set of seven fixed patterns have been taken into account. Based on the obtained results in these initial experiments we expect a reasonable performance for more realistic scenarios. Together with expanding the experiments for the exact and partial matching algorithm, we expect to carry out a set of experiments for the generalised and hierarchical pattern matching algorithms - only outlined in this paper.

6. Related work

A number of publications have addressed the problem of service discovery, service design, process model comparison and querying process descriptions. They differ in scope, focus and the medium to reach their objectives. Service discovery is close to our approach in cases where service requests are defined in the form of process centric descriptions and they are matched against descriptions of available services, such as in [21],[8],[10]. Pattern-based service design methodologies provide a context to our approach, and when automation is a core concept, our solutions can play an important role. Solutions to compare and to query process models are close to our pattern matching solutions. Because we focus on a business process pattern centric service design, solutions helping to find pattern occurrences in process models are relevant related work, for example the work presented in [2],[4],[6].

In [8] a technique for partial matching on behavioral models is presented. The proposal provides measures of semantic distance between resultant matches and user requirements. Several issues regarding complexity of the proposed algorithm are reported to be improved. However, experimental results indicate a response time of approximately thirty seconds for a target graph of fifty vertices, which can be prohibitive for large processes. In [4] a method to measure structural distance between process definitions associated to web services is presented. The method relies on a distance measure of normalised matrices representing graph-based process models. Improvements on the data structure for matrices could provide more flexibility to represent processes and improve performance. In [10] various types of structural matches for BPEL processes supporting dynamic binding of services are defined. BPEL processes are modelled as process trees, where each tree node is an interaction. Activities which are not interactions are abstracted into internal steps and can not be matched. Duplicate interaction activities are not allowed in the tree. *Plugin* matching is presented as an approach based on a process simulation notion, however such as the authors indicate, the proposal requires further semantic analysis to decide if a process can replace another after a matching. In [6], the authors propose a query language for BPEL process descriptions based on Context Free Graph Grammars (CFGG) - which in general are not closed under intersection. Replacement in the considered CFGG involves isomorphic relations between graphs. In our approach, structural relations between processes and patterns (queries) involves surjective graph homomorphisms. In [6], process queries are graphical queries annotated in the same way as process descriptions. Activities can be zoomed-in by means of graph refinement. Cycles in process graphs and during graph refinements containing recursion are handled by representing compacted graph structures. Many fork and joins constructs could lead to a exponential number of paths in query's results. Labels in a query and a query answer require syntactical equivalence. Extensions to consider label predicates and regular path expressions are discussed. In [2] the authors propose a way to compare two process models based on their observed behavior. Observed behavior relies on the information extracted from logs of process executions. Mining techniques are applied over sequences of process steps. Our focus is rather on graphs representing process models, this might include the results of mining techniques that obtain graph-based models representing sets of process executions.

7. Conclusion

In this paper we have discussed the benefits, the concerns and some possible solutions to automatically recommend the scope and granularity of services based on identified patterns in business process models. At the core, the approach uses a set of graph matching algorithms. We discussed some concerns and proposed solutions for exact, inexact, partial, generalised and hierarchical pattern matching. These includes semantic matches beyond syntactic equivalence and consideration of matches at different level of abstraction in process models.

Additionally, we proposed a solution to discover frequent patterns -named utility patterns- in process models. Utility patterns, together with an appropriate traceability

support relating software components to process steps, can provide recommendations to define the scope and granularity of reusable technical-centered services.

Our initial motivation in this work was based on the potential benefits that pattern matching and pattern discovery techniques could provide to business analysts and architects during the definition of new process-centric services. Process models can be annotated with the results of the pattern matching and presented to designers on standard modelling tools. Note that in this paper we have assumed the availability of process models and/or process-centric service descriptions and their related patterns. In current real scenarios, the availability of process documentation might be considered as low. However, we believe that business and architectural documentation in the form of process-centric models is becoming more and more relevant in the context of service architecture implementations and public workflows, for example in the grid workflow environment.

Models documenting real case scenarios are complex, numerous and often large. Automation is core to improve effectiveness and efficiency during the analysis of these models. Our proposal aims to support designers by automating some of the steps during the analysis and design of process-centric service architectures descriptions.

We believe that architecture abstractions, such as patterns, are a powerful concept that can be exploited to improve the design of new services. Further work regarding performance and scalability of our proposed techniques is in development. We plan to investigate their applicability to dynamic service composition.

References

[1] Aalst, W.M.P. van der, Hofstede, A.H.M. ter, Kiepuszewski, B., Barros, A. P. *Workflow Patterns*. Distributed and Parallel Databases **14**(1):5–51. (2003).

[2] Aalst, W.M.P. van der, Alves de Medeiros, A.K., Weijters, A.J.M.M. *Process Equivalence : Comparing Two Process Models Based on Observed Behavior*. In: S. Dustdar, J.L. Fiadeiro, A. Sheth (Eds.), Business Process Management (BPM'06). Springer, LNCS, Vol. 4102, pp. 129-144 (2006).

[3] Abramovsky, L. and Griffith, R., *Outsourcing and Offshoring of Business Services: How Important is ICT?*. Journal of the European Economic Association **4**(2-3):594–601, MIT Press (2006).

[4] Bae, J., Liu, L., Caverlee, J., Rouse, W.B., *Process Mining, Discovery, and Integration using Distance Measures*. In: Proc. IEEE International Conference on Web Services (ICWS'06), IEEE Computer Society, p. 479-488, (2006).

[5] Barros O., *Business Process Patterns and Frameworks: Reusing Knowledge in Process Innovation*. Business Process Management Journal **13**(1):47-69, Emerald Group (2007)

[6] Beeri, C., Eyal, A., Kamenkovich, S., Milo, T., *Querying Business Processes with BP-QL*. Information Systems 33(6):477-507, Elsevier (2008)

[7] Buschmann, F., Henney, K., Schmidt, D.C., *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. 1st Edition, Wiley & Sons (2007).

[8] Corrales, J., Grigori, D., Bouzeghoub, M., *BPEL Processes Matchmaking for Service Discovery*. In: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, p.237-254, Springer (2006).

[9] Erl, T., *Service-oriented architecture: Concepts, Technology, and Design*. Prentice Hall (2004).

[10] Eshuis, R., Grefen, P., *Structural Matching of BPEL Processes*. In: 5th European Conference on Web Services (ECOWS'07), p. 171-180, IEEE Computer Society, (2007).

[11] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G., *Handbook of Graph Grammars and Computing by Graph Transformation, Part II: Applications, Languages and Tools*. World Scientific (1999).

[12] Fettke, P., Loos, P., *Reference Modeling for Business Systems Analysis*. IGI Publishing (2006).

[13] Fiala, J., *Structure And Complexity of Locally Constrained Graph Homomorphisms*. PhD Thesis, Charles University, Faculty of Mathematics And Physics (2007).

[14] Fiala, J., Kratochvil, J., *Locally constrained graph homomorphisms–structure, complexity, and applications*. Computer Science Review **2**(2):97–111, Elsevier Inc. (2008).

[15] Gschwind T., Koehler J., Wong J., *Applying Patterns during Business Process Modeling*. In: Dumas M., Reichert M., Shan M.-C. (Eds.), 6th International Conference on Business Process Management (BPM'08), Springer, LNCS, Vol. 5240, p. 4-19 (2008).

[16] Hell, P., Nesetril, J., *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and Its Applications, Oxford University Press, Vol. 28 (2004).

[17] Kuramochi, M., Karypis, G., *Finding Frequent Patterns in a Large Sparse Graph*. Data Mining and Knowledge Discovery **11**(3):243–271, Kluwer Academic Publishers (2005).

[18] Sadiq, W., Orlowska, M.E., *Analyzing process models using graph reduction techniques*. Information Systems **25**(2):117–134, Elsevier Science Ltd (2000).

[19] Tong, H., Faloutsos, C., Gallagher, B., Eliassi-Rad, T., *Fast best-effort pattern matching in large attributed graphs*. In: Proc. 13th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD'07), P. 737–746, ACM (2007).

[20] Vanhatalo J., Vlzer H., Koehler J., *The Refined Process Structure Tree*. Business Process Management (BPM'08), Springer, LNCS, Vol. 5240, p.100-115 (2008)

[21] Wombacher, A., Rozie, M., *Evaluation of Workflow Similarity Measures in Service Discovery*. In: Schoop, M., Huemer, C., Rebstock, M., Bichler, M. (Eds.) Service Oriented Electronic Commerce, GI, Vol(80), p.51-71 (2006).

8. Annex: Graphs

This annex is based on Nešetřil, Fiala and Hell's work [16],[14], [13].
A ***graph*** G is a set V_G of vertices together with a set E_G of edges, where each edge is a two-element set of vertices. If V_G is finite, the graph G is called a ***finite*** graph. If the graph has orientation, it is called ***directed*** graph, and each edge is called an *arc*. An arc can have one of the two orientations (u,v) or (v,u) with $u,v \in V_G$. If loops on vertices are allowed, then edges consist of only one vertex, written (u,u) with $u \in V_G$. A sequence of vertices of a graph G, such that the consecutive pairs are adjacent, is called a *walk* in G. If all vertices in a walk are distinct, then it is called a *path*. A graph G is called a ***connected*** graph if for every pair of vertices $u,v \in V_G$ there exists a finite path starting in u and ending in v.

For a vertex u in a graph G, the set of all vertices adjacent to u are called the ***neighborhood*** of u and is denoted by $N_G(u)$, with $N_G(u) = \{v|(u,v) \in E_G\}$. Consequently, a vertex v is a neighbor of u if u and v are adjacent. A graph G is a ***subgraph*** of H if $V_G \subseteq V_H$ and $E_G \subseteq E_H$.

Homomorphisms. Graph homomorphisms are edge preserving vertex mapping between two graphs. A *graph homomorphism* from G to H denoted by $G \rightarrow H$ is a vertex mapping $f : V_G \rightarrow V_H$ satisfying $(f(u), f(v)) \in E_H$ for any edge $(u,v) \in E_G$. According to [14], whenever a homomorphism $G \rightarrow H$ holds, then the image of the neighborhood of a vertex from the source graph V_G is contained in the neighborhood of the image of that vertex in the target graph V_H, i.e. $f(N_G(u)) \subseteq N_H(f(u))$ for all $u \in V_G$. Composition of two homomorphisms $f : F \rightarrow G$ and $g : G \rightarrow H$ is another homomorphism $g \circ f : F \rightarrow H$. If a homomorphism $f : G \rightarrow H$ is an one-to-one mapping and f^{-1} is also a homomorphism, then f is called an ***isomorphism***. In such a case is said that G and H are isomorphic and it is denoted by $G \simeq H$. An isomorphism $f : G \rightarrow G$ is called an automorphism of G, and the set of all automorphisms of G is denoted by $AUT(G)$.

Using the latter notation, for graphs G and H three kind of homomorphic mapping are defined as:

- $G \xrightarrow{B} H$ if there exist a *locally bijective homomorphism* $f : V_G \rightarrow V_H$ that satisfies for all $u \in V_G : u \in V_G : f(N_G(u)) = N_H(f(u))$ and $|f(N_G(u))| = |N_G(u)|$.
- $G \xrightarrow{I} H$ if there exist a *locally injective homomorphism* $f : V_G \rightarrow V_H$ that satisfies for all $u \in V_G : |f(N_G(u))| = |N_G(u)|$.
- $G \xrightarrow{S} H$ if there exist a *locally surjective homomorphism* $f : V_G \rightarrow V_H$ that satisfies for all $u \in V_G : f(N_G(u)) = N_H(f(u))$.

Note that for the mappings above, locally bijective homomorphism is both locally injective and surjective. The mappings are also known in the literature as (full) covering projections (bijective), or as partial covering projections (injective), or as role assignments (surjective). Additionally, any locally surjective homomorphism f from a graph G to a connected graph H is globally surjective, and any locally injective homomorphism f from a connected graph G to a forest H is globally injective [13].

Labelled Graphs. The graph $G = (V_G, E_G, \ell_{V_G}, \ell_{E_G})$ is a graph where the vertices in V_G and edges in E_G have labels. The functions assigning labels to vertices and edges are surjective homomorphisms $\ell_{V_G} : V_G \rightarrow L_{V_G}$ and $\ell_{E_G} : E_G \rightarrow L_{E_G}$ for all the vertices in V_G and the edges in E_G, respectively. L_{V_G} and L_{E_G} are the sets of vertex labels and edge labels, respectively. Note that surjection allow the existence of a same label in $L_{V_G}(L_{E_G})$ for several vertices(edges).

Acknowledgment

We want to thanks Lero - The Irish Software Engineering Research Centre and CONICYT (Chile) for supporting this work.

Veronica Gacitua-Decar and Claus Pahl
Lero, School of Computing
Dublin City University
Glasnevin, Dublin 9
Ireland
e-mail: vgacitua|cpahl@computing.dcu.ie