Lifting Low-Level Workflow Changes through User-Defined Graph-Rule-Based Patterns

Alexander Jahl, Harun Baraki, Huu Tam Tran,
Ramaprasad Kuppili, and Kurt Geihs

Distributed Systems Group
University of Kassel
Kassel, Germany
{jahl, baraki, tran, rkuppili, geihs}@vs.uni-kassel.de

Abstract. In dynamic service-oriented architectures, services and service compositions underlie constant evolution that may not only affect the own workflow but dependent services too. Subsequently required adaptations necessitate an effective detection of the changes and their effects. Merely capturing a sequence of low-level changes and analyzing each of them demands much coordination and may lead to an incomplete picture. An abstraction that summarizes a combination of low-level changes will facilitate the detection and reduce the number of changes that shall be considered for adaptation. In this paper we propose an abstraction that is formulated through graph-based patterns, since service compositions are workflows that can be mapped to directed labeled graphs. The characteristics and granularity of a graph pattern can be adjusted by domain experts to the respective workflow language and application case. In particular, graph-based patterns are crucial when workflows are represented in two different formats. This could be the case if there exists one representation for the execution and one for the verification. We present implementation details and a detailed example that show the feasibility and simplicity of our solution.

Keywords: Graph Transformation, Graph Matching, Pattern Matching, Change Impact Analysis, Dependency Graph, Web Services, Service Evolution, Answer Set Programming

1 Introduction

It is common wisdom that actively used software must be evolved continuously in order to maintain its utility and quality [12]. Adding new features, removing obsolete features, fixing bugs, closing security holes, improving performance all require updating a software product from time to time. Certainly, this is also true for services provided via a computer network. However, services seldom work in isolation in a stand-alone fashion. They may be part of business processes where services depend on other services and may be composed of other services.

These manifold interdependencies make on-the-fly service evolution a particularly difficult and challenging problem because the evolution of one service may

incur changes in other dependent services and clients. In analogy to biology, we call this service co-evolution. Our goal is to provide a general solution for coordinated decentralized service co-evolution. Such a solution is lacking. For heavily used services in business-critical application scenarios upgrade-related downtime is not acceptable in most cases, but often the reality. Hence, on-the-fly, zero downtime service co-evolution is a major objective for our research.

This demands, first and foremost, an examination of the effects and consequences of each alteration in a formalized way. Current formal specification methods for Change Impact Analysis (CIA) apply logic programs, state machines, and semantic annotations. In [4] compositions are formulated in Prolog. This enables developers to perform consistency checks through atomic Prolog queries. Ryu et al. [15] map protocols of service compositions to finite state machines. After a change occurs, the protocol compatibility of the participating services will be tested to decide about a migration to the updated protocol version. Likewise, other works in this area also focus on updates which encompass a single removal or addition of a service or parameter, check the consistency after the update or inform affected parties [1]. However, the consideration of single update steps impedes detecting and processing complex changes like replacements, swaps, or the addition of new branches and subgraphs to the workflow. These changes are still treated as a sequence of low-level updates. Lifting a sequence of low-level changes to a complex change captures additional information that would be lost otherwise. By lifting we refer to the conversion of low-level changes to a more abstract, conceptional description of model modifications. A sequence of removals and additions of services at the same position would not be interpreted as a replacement. The detection of a complex change would allow triggering tailored actions for handling and checking them. In case of a replacement, the pre- and post-conditions of the replaced and the replacing service could be checked for compliance.

This work provides a practical solution for developers to define complex change patterns by means of a simply applicable graphical approach. The change patterns are formulated in terms of hierarchically organized graph rules. This enables identifying and categorizing changes with different granularities. Hence, nested rules are feasible that trigger actions for each level of the hierarchy, e.g. log removal and insertion and verify replacement. Just considering detected higher level changes, e.g. the replacement, would lead to a comprehensive view that compacts dealing with changes. Further applications, which will not be discussed in this work, include the storage and the communication and dissemination of changes in a flexible and dense format.

In this paper, we present our change pattern definition and detection approach. Both are implemented by means of our DiCORE:CIA (Distributed Cooperative Evolution: Change Impact Analysis) module. Additionally, we extend this module by an Answer-Set-Programming-based logic programming reasoner to verify the updated model and to draw conclusions about dependent components. Throughout the paper a comprehensive example is used consistently to illustrate the course of action and demonstrate the practicability of our solution.

The remainder of this paper is organized as follows. **Section II** introduces our graph-rule-based change patterns and justifies the need for them. Thereafter, **Section III** presents the architecture and functionality of our first DiCORE:CIA prototype. Related work is discussed in **Section IV**. Finally, the main findings of this paper and future work are summarized in **Section V**.

2 Graph-rule-based Patterns

We will start with the illustration of a service workflow that is changed and extended by the responsible service provider. With the aid of this running example, the subsequent sections demonstrate the definition and viability of our graph-rule-based patterns (GPs).

2.1 Scenario

The workflow depicted in Fig. 1 is an orchestration that can be executed on a client or service provider machine. Workflows usually encompass different types of nodes which are commonly termed *activities*. An activity may be the invocation of a local or remote service and may also encompass user interaction. Workflow WF1 includes one remote service invocation. The *getMap* function of service *S1* is requested. The other operations are executed locally. *choosePOI* and *getGPScoord* enable a user to select a Point of Interest and acquire its GPS coordinates. These are processed remotely by service *S1* that returns a detailed map of the corresponding area. The following activities create a suitable route plan (pedestrian, car, public transport, bicycle) and display it accordingly. Now let us assume that a workflow update provides a more precise localization and an additional map type is offered. This new workflow WF2 is shown in the lower part of Fig. 1. Additionally, service *S1* is replaced by service *S2*. The pollution map in workflow WF2 allows users to choose routes with low emission levels.

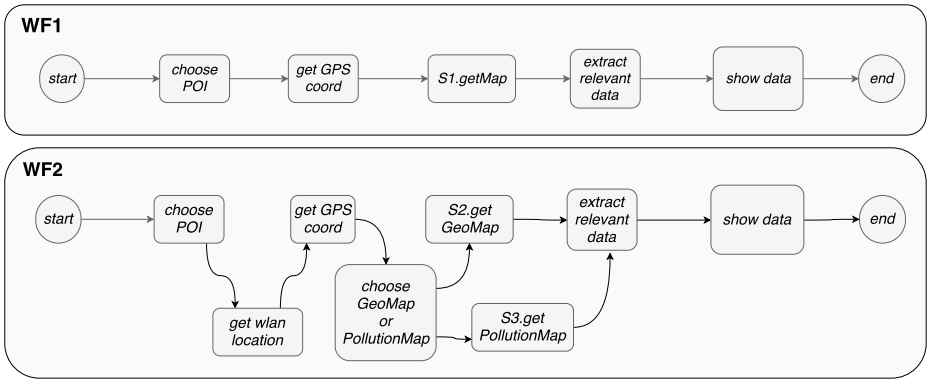

Fig. 1. Scenario: original workflow WF1 and revision WF2

Note that the workflows WF1 and WF2 in Fig. 1 are represented by a directed graph. This restricts the search area for the following graph matching approach significantly, increases its accuracy and reduces its faults. Nevertheless, it should be pointed out here that our GPs are applicable for undirected graphs, too.

2.2 Graph Comparison and Reduction

Before explaining GPs and their application in detail, our graph comparison and reduction approach is presented. The graph comparison matches nodes of two subsequent versions of a workflow to detect atomic changes, more precisely, it generates a list of added and removed nodes. The graph reduction is required to consider the addition or removal of subgraphs as the addition or removal of one node. This enables, for instance, that the replacement of a node by a subgraph is detected correctly.

Graph Comparison Subgraph matching is the challenge to find all matches of a query graph. This task is known to be an NP-complete problem. A comparison between two graphs G_1 and G_2, where:

- $G = (N, E)$,
- N is a set of nodes,
- E is a set of edges,

consists in the determination of a mapping M that associates nodes of G_1 with nodes of G_2 in compliance with some predefined constraints. The mapping M is represented by a set of pairs $(m \in G_1, n \in G_2)$, each pair representing the mapping of a node m from G_1 with a node n from G_2. A mapping $M \subset N_1 \times N_2$ is an isomorphism if M is a bijective function that preserves the branch structure of the two graphs. For comparing the two versions of a graph, we compute the minimal graph edit distance [6] between them by using an A* algorithm and calculating the string edit distance (syntactical analysis) [13] for a structural matching of each corresponding graph node pair. Additionally, a semantic analysis [9] calculates a degree of similarity based on the equivalence between the words they consist of.

In our graph comparison algorithm depicted in Algo. 1, this functionality is implemented by the *searchEquivalent* function listed in line 2. The algorithm starts with a start node N of the source graph and returns by use of *searchEquivalent* the best match as *equivalentN*. The match precision is obtained through an invocation of *getPercentFor*. The matching and its similarity assessment are stored into M as a triple. The *compare* function is invoked recursively for neighbour nodes found through direct edges (line 7). Finally, M contains the best matches between node $n \in N_1$ of a source graph G_1 and node $m \in N_2$ of the target graph G_2. Matches whose similarities fall below a threshold, will be discarded in a subsequent step. Applying the graph comparison method to our scenario in Fig.1 results in a pairwise mapping of corresponding nodes of the source and the target graph. The outcome of the comparison depicted in Fig. 3

Algorithm 1: Subgraph comparison

```
1  function compare (N, G);
   Input  : N, start node from source graph; G, target graph
   Output: M, mapping result
2  equivalentN = searchEquivalent(N, G);
3  percent = getPercentFor(N, G);
4  if equivalentN then
5  |  M.put (N, equivalentN, percent);
6  |  foreach edge in N.edges do
7  |  |  nodes = edge.nodes;
8  |  |  foreach node in nodes do
9  |  |  |  if node != N then
10 |  |  |  |  M.putall(compare (node, G));
11 |  |  |  end
12 |  |  end
13 |  end
14 end
15 return M;
```

reveals that not all nodes have compliant counterparts. These nodes are marked **Green** when added and **Red** when removed (Fig. 3). The graph matching and the marked nodes form the starting points for the application of our change patterns.

Graph Reduction Preparatory steps that summarize the addition or removal of directly connected nodes will improve the scalability of our graph matching by reducing the search space. Besides that, the definition and application of GPs will be simplified. For instance, a developer may create a replacement GP by means of a removed and inserted node at the same position. A single node that is replaced by a subgraph would be identified by this GP because the insertion

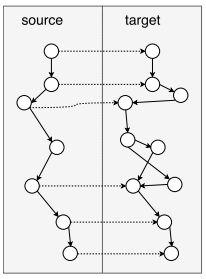

Fig. 2. Matching of Source graph G_1 nodes and the corresponding target graph G_2 nodes

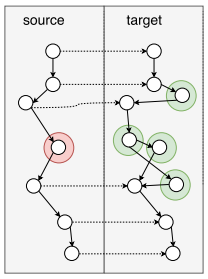

Fig. 3. Graph matching with marked added and deleted nodes

of the subgraph is reduced by our approach to an insertion of one single node. The two middle columns in Fig. 4 illustrate this amalgamation. Contracted nodes and their connections will be saved as subgraphs that are analyzed in subsequent steps.

A subgraph H is defined as a subset of vertices and edges of a graph $G = (N, E)$, with $H = (N_H, E_H, \mu, \nu, L_\mu, L_\nu)$ where:

- $N_H \subseteq N$,
- $E_H = E \cap (N_H \times N_H)$,
- μ function matching label $l \in L_\mu$ to node $n \in N_H$,
- ν function matching label $l \in L_\nu$ to edge $e \in E_H$,
- $L_\mu = L_\nu = \{x | x = green\} \ \oplus \{x | x = red\}$.

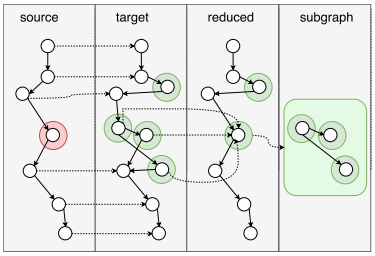

Fig. 4. Identify subgraph of new added nodes and contract to one node

2.3 Graph-rule-based Patterns

Our GPs are formulated by the usage of graph rules that are organized hierarchically and which enable identifying changes with different granularities. The GP notation is based on the graph rule syntax of [10, 5].

Graph Rules (GRs) are undirected labeled graphs defined as a 6-tuple $R = (N, E, \mu, \nu, L_\mu, L_\nu)$ where:

- N is a set of nodes,
- E is a set of edges,
- μ function matching label $l \in L_\mu$ to node $n \in N$,
- ν function matching label $l \in L_\nu$ to edge $e \in E$,
- L_μ is a set of symbolic labels to mark nodes.
- L_ν is a set of symbolic labels to mark edges.

In general, graph rules can be used to transform a graph from one domain to another. Therefore, they connect corresponding nodes and edges and mark new and deleted components. In our case, the source and target domain of our GRs

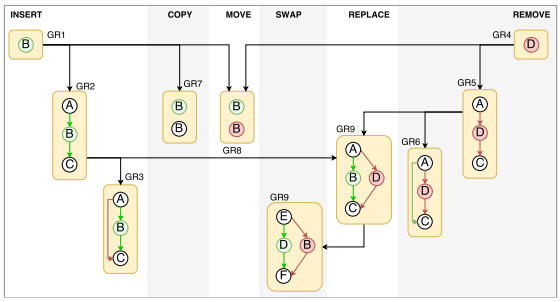

Fig. 5. Example set of graph rules to detect low- and high-level changes

may be equal. A GR combines and includes information about added and deleted nodes and edges, e.g. in parallel or as a new branch, and their context.

GRs are formulated through a graphical representation. Fig. 5 presents an example set of GRs and their interdependencies. GR1 stands for the insertion of a single node. Further information is gained, if, for instance, subsequently GR2 and GR3 can be matched. This represents the replacement of a direct connection between a node A and C by a node B and its connections to A and C.

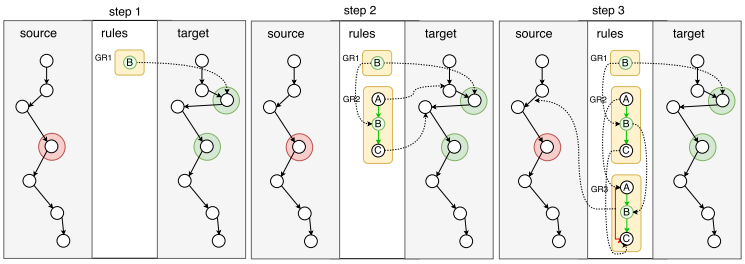

Fig. 6. Example of the application of successive and interrelated graph rules to detect the kind of change as precisely as possible

The application of this GR set to our example scenario is illustrated in Fig. 6. After the execution of the aforementioned graph matching (Figures 2, 3 and 4), GR1 is detected in the first step. This can be interpreted programmatically as an added node. In the second step, further GRs reachable from GR1 are tried to be matched. Here, GR2 is fired which indicates an insertion between two existing nodes. Finally, GR3 identifies that these two existing nodes were directly connected before and, hence, the direct connection was replaced. Fig. 7 and Fig. 8 demonstrate the described procedure or the other two identified nodes.

The colors in these GRs are part of the syntax. In our case, L_μ and L_ν have the same set of labels. We define L_μ and L_ν as follows:

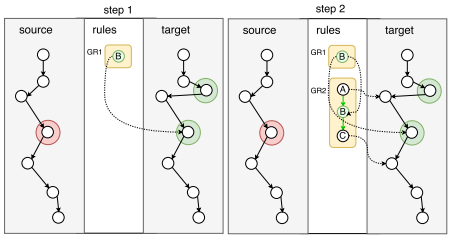

Fig. 7. Detection added node (step 1), identification neighbour nodes (step 2)

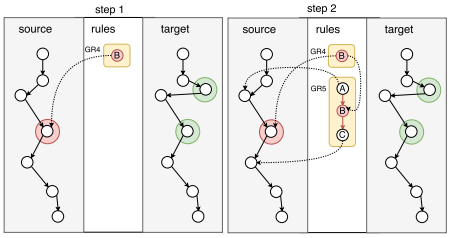

Fig. 8. Detection removed node (step 1), identification neighbour nodes (step 2)

– $L_\mu = L_\nu = \{black, green, red\}$.

They indicate which elements are added, removed or remained unchanged. Elements marked **Black** are used to find corresponding elements in both graphs. Hence, they serve as context. **Green** elements were added in the target graph and do not exist in the source graph at the same position. **Red** elements exist in the source graph but do not exist at that position in the target graph. In the depicted examples, node names refer to the corresponding functionality, that is, same names stand for the same functionality type.

GR-based Change Patterns GPs can be considered as a combination of GRs. A GP may stand for a specific statement. For instance, GR1 to GR3 are subsumed as an *Insert GP*. Formally, a GP is a directed graph that contains GRs as nodes:

– $P = (N_R, E)$
– N_R is a set of R,
– E is a set of edges, formulated as $E \subseteq N_R \times N_R$.

GPs can build on one another so that more details and information are captured. Linking GPs means to share related GRs. Fig. 9 demonstrates this by connecting the *Insert GP* and *Remove GP* with the *Replace GP*. Coming back to our scenario, let us assume that a developer or user replaced an existing node D by a new node B. In Fig. 9, node B represents a subgraph. Initially, GR1 and GR4 match the added and removed nodes. These are included in the *Insert GP* and *Remove GP*. Subsequently, GR2 and GR5 identify the context nodes. GR3 will not be activated in this case since there was no direct connection between node A and C beforehand. The *Replace GP* is confirmed only if both GR2 and GR5 fire with the same context nodes A and C. The bidirectional arrows in the *Replace GP* indicate that it cannot be valid if merely one GR matches. Starting with simple rules, developers can extend them by adding GRs and GPs in a hierarchical manner to detect more complex changes. This allows, inter alia, the recognition of patterns like swaps, parallelizations and other domain specific structures.

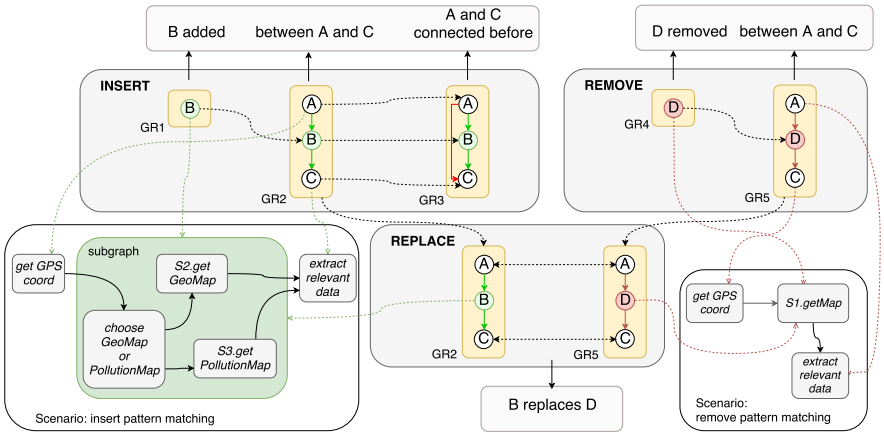

Fig. 9. GR-based change patterns *Insert, Remove* and *Replace*

The hierarchical GRs shown in Figure 9 specify the search path for each change pattern. Starting at node GR1 or GR4, the algorithm presented in Algo. 2 checks after a successful matching if one of the following rules can be applied. In our example, that would be GR2 or GR5. If a GR matches, all relevant information like context nodes or added or removed edges are set at that point. Following GRs will use the same context information. Our match checking ap-

Algorithm 2: GP graph matching

```
1 function findMatching (G, R);
  Input  : G, graph with included change marker; R, current rule from GP
  Output: A, map of all matched rules and connected G nodes
2 match = checkMatch(G, R);
3 if match then
4 |   A.put(R, match);
5 |   foreach childR in R.children do
6 |   |   A.put( findMatching(G, childR) );
7 |   end
8 end
9 return A;
```

plies exact subgraph isomorphism to resolve this kind of pattern matching. Any successful executed graph rule provides new and additional informations.

Answer Set Programming follows the Declarative Programming paradigm and has its roots in logic programming, non-monotonic reasoning, and databases. It is used for planning and diagnosis of NP-hard search problems. ASP provides

the possibility of simple ASP code generation in a readable format and reasoning about temporal and structural dependencies in a workflow. A detailed explanation can be found in [8].

Our framework translates process graphs to ASP in order to verify locally the workflow after changes are performed. If a consistency violation is detected, the change is discarded and the developer is notified. If changes affect the in- or output of a workflow, they may affect clients using this workflow. Since in- and outputs of workflows are formulated in ASP too, they serve as change description for these affected clients. Our DiCORE framework is also running on client side. This allows us to receive and process ASP fragments and check for consistency on the client side. Consistency violations will cause a service replacement so that pre- and post-conditions of each activity on client side are fulfilled. Fig. 10 shows the transition diagram of a workflow including two Web service invocations. Transition diagrams can be directly translated to ASP [8]. The ASP translation of the transition diagram is given in List. 1.1.

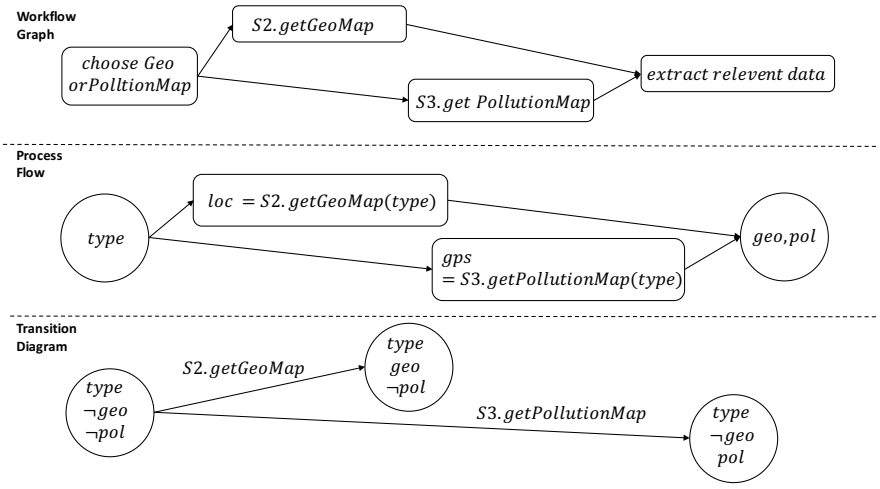

Fig. 10. Process Graph to ASP

The translation to ASP is only executed for the original workflow. Whenever changes occur in the workflow graph, they will be detected, condensed and mapped to GPs. Each GP holds a corresponding translation in ASP which is generated automatically during the GP creation phase. This ensures that ASP descriptions can be updated instead of being generated anew.

Assuming that *S2* and *S3* are inserted into an existing workflow, our Graph Reduction would summarize this event as an insertion of a subgraph. This simplification step is also reflected automatically in ASP (List. 1.2). It results in a shortened ASP description which reduces the search space for valid models. Eventually, the subgraph can be resolved into its original composition.

Listing 1.1. ASP translation

```
fluent(inertial,type)
fluent(inertial,geo)
fluent(inertial,pol)

action(getGeoMap)
action(getPollutionMap)

holds(geo, t+1) :- holds(
    type, t), holds(-geo, t
    ),holds(-pol, t),
    occurs(getGeoMap, t)
holds(pol, t+1) :- holds(
    type, t), holds(-geo, t
    ),holds(-pol, t),
    occurs(getPollutionMap,
     t)

holds(type, 0)
holds(-geo,0)
holds(-pol,0)
occurs(getGeoMap,0)
occurs(getPollutionMap,0)
```

Listing 1.2. ASP after reduction

```
fluent(intertial,type)
fluent(intertial,m)

action(Subgraph#getMap)

holds(m, t+1) :- holds(type,
     t), holds(-m, t),
    occurs(Subgraph#getMap,
     t)

holds(type, 0)
holds(-m,0)
occurs(Subgraph#getMap,0)
```

3 DiCORE:CIA

The DiCORE framework determines the kind of changes and the affected components in a business process and communicates them with dependent clients. DiCORE:CIA detects functional changes by analyzing the structure of the workflow. This module is implemented as a Java library and is part of our DiCORE framework. In combination with our ASP component (based on clingo [7]), it supports the developer conveniently through assistance for recognizing changes and their consequences. Furthermore, a graphical editor for visualizing, customizing and extending the GPs to particular requirements is provided. The following sections present an overview of the DiCORE:CIA module architecture and explains the main features.

3.1 Architecture overview

DiCORE:CIA includes the following four packages: (1) Process Graph Converter, (2) Graph Matching, (3) Graph UI, (4) ASP Analyzer.

The Process Graph Converter uses a data model, imported from one of the well-known workflow languages, e.g. UML, BPMN, BPEL, YAWL, or EPC, to generate a process graph, based on the process graph syntax in [3].

The Graph Matching package consists of three main components: (1) Graph Comparator, (2) Graph Reduction, (3) Pattern Matching. The Graph Comparator matches two versions of a process graph and extracts the differences. The Graph Reduction component contracts the outputs of the Graph Comparator. Finally, the Pattern Matching component applies the GR graph to match GPs.

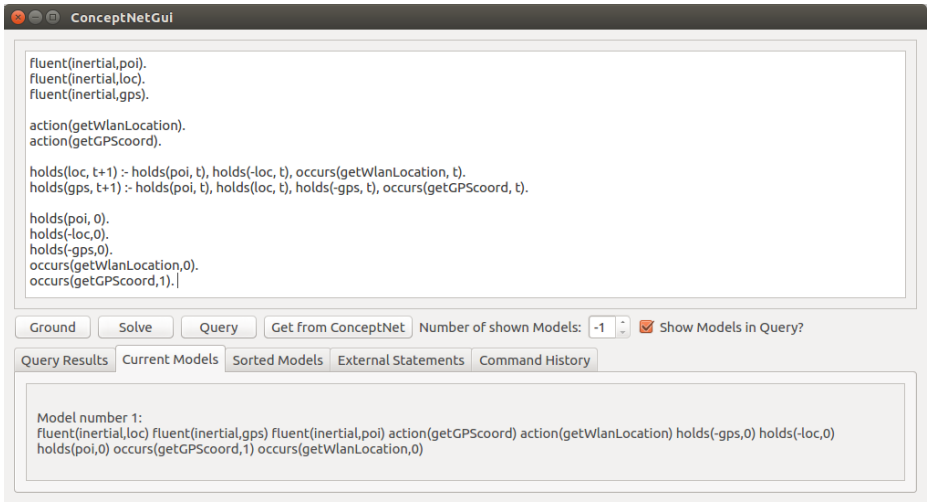

Fig. 11. ASP Viewer [14]

The Graph UI package encompasses three components: (1) DiCORE GR Editor, (2) DiCORE Visualization, (3) ASP Viewer. The DiCORE GR Editor is the UI interface for designing the graph-based change patterns. The implementation is based on JavaFX[1]. The graphical interface provides functions for creating and editing hierarchically structured GRs as well as tools to import them from and export them to JSON and XML. DiCORE Visualization shows live information about process graphs and their changes. The ASP Viewer depicted in Fig. 11 shows the corresponding ASP models and allows further analytical steps not discussed in this work.

The ASP Analyzer contains three components: (1) ASP Converter, (2) ASP Query Generator, (3) ASP Updater. The process graph serves as input for the ASP Converter which generates the ASP description needed for further analytical steps. The ASP Query Generator executes consistency checks through automatically created queries and identifies after performed changes affected nodes inside the process graph.

4 Related work

Workflows are always prone to different kinds of changes, such as new regulatory laws, changes in policies or strategies, or emerging technologies. Therefore, detecting, logging and notifying about functional changes in processes is a critical step in change analysis. This section will present a brief review of research work dealing with the detection and handling of functional changes.

[1] Standard GUI library for Java

For detecting functional changes, various works applied comparisons between processes to extract different kind of information. These, however, do not provide tools to developers to define their own patterns or change types, but consider a fixed set of possible atomic changes.

Aamas et. al [2] propose the Bp-diff tool for a comparison of business models. The Bp-Diff tool may identify discrepancies involving pairs of tasks and provides both textual and visual feedback to help users understand each discrepancy. The textual feedback explains how a given pair of tasks is related in the given model versions. The visual feedback allows users to pinpoint the exact state where the discrepancy occurs. Similarly, Sergey et al. [11] present their tool BPMNDiffViz which compares process models represented in BPMN. The authors provide a web-based tool that finds business process discrepancies and visualizes them. However, this tool is applied only for BPMN formats and only considers atomic changes.

Further alternative approaches are based on predefined sets of change patterns. The authors in [16] suggest a set of change patterns like the addition and removal of process fragments or moving or replacing fragments. Our approach encompasses these patterns and is additionally editable and extendable through our GPs.

5 Conclusions

This paper presents a new contribution for formulating and detecting user-defined change patterns in complex service graphs. The patterns can be structured hierarchically and allow a categorization of changes. The hierarchic structure enables to capture additional information with each GP that could be matched to a change. Furthermore, our solution allows an intuitive and graphical formulation of patterns while other existing tools completely ignore user-defined change patterns. Our DiCORE framework presents a first implementation of the GPs. It communicates workflow changes with affected parties and triggers adaptations in case of inconsistencies. Therefore, we employ logic programming by automatically generated ASP descriptions that are processed by an ASP solver. GPs are not restricted to this application scenario but can be applied in general for detection, compression, logging and communication of simple and complex changes in a graph-based description expressed as interconnected graph rules.

In a future work, a special multi-agent system shall communicate the changes and coordinate possible adaptations with affected parties. Therefore we extend our DiCORE framework and continue to promote this research area.

Acknowledgment

This work is supported by the German Research Foundation (DFG) under the project PROSECCO, grant number 5534111. The authors would like to thank the DFG for supporting their participation in worldwide research networks.

References

1. Alam, K.A., Ahmad, R., Akhunzada, A., Nasir, M.H.N.M., Khan, S.U.: Impact analysis and change propagation in service-oriented enterprises: A systematic review. Information Systems 54, 43–73 (2015)
2. Armas-Cervantes, A., Baldan, P., Dumas, M., Garcıa-Banuelos, L.: Bp-diff: A tool for behavioral comparison of business process models. Proceedings of the BPM Demo Sessions pp. 1–6 (2014)
3. Bouchaala, O., Yangui, M., Tata, S., Jmaiel, M.: Dat: Dependency analysis tool for service based business processes. In: 2014 IEEE 28th International Conference on Advanced Information Networking and Applications. pp. 621–628. IEEE (2014)
4. Dai, W., Covvey, D., Alencar, P., Cowan, D.: Lightweight query-based analysis of workflow process dependencies. Journal of Systems and Software 82(6), 915–931 (2009)
5. Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G.: Graph Transformations. Springer (2008)
6. Fischer, A., Suen, C.Y., Frinken, V., Riesen, K., Bunke, H.: Approximation of graph edit distance based on Hausdorff matching. Pattern recognition 48(2), 331–343 (2015)
7. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with Clingo 5. In: OASIcs-OpenAccess Series in Informatics. vol. 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2016)
8. Gelfond, M., Kahl, Y.: Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach. Cambridge University Press (2014)
9. Gomaa, W.H., Fahmy, A.A.: A survey of text similarity approaches. International Journal of Computer Applications 68(13) (2013)
10. Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., Varro, D.: Using graph transformation for practical model-driven software engineering. In: Model-driven Software Development, pp. 91–117. Springer (2005)
11. Ivanov, S., Kalenkova, A., van der Aalst, W.M.: BPMNDiffViz: A Tool for BPMN Models Comparison. In: BPM (Demos). pp. 35–39 (2015)
12. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68(9), 1060–1076 (1980)
13. Lu, W., Du, X., Hadjieleftheriou, M., Ooi, B.C.: Efficiently Supporting Edit Distance Based String Similarity Search Using B+ -Trees. IEEE Transactions on Knowledge and Data Engineering 26(12), 2983–2996 (2014)
14. Opfer, S., Jakob, S., Geihs, K.: Reasoning for Autonomous Agents in Dynamic Domains. Agents and Artificial Intelligence, 9th International Conference, ICAART 2017 (2017)
15. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R.: Supporting the dynamic evolution of web service protocols in service-oriented architectures. ACM Transactions on the Web (TWEB) 2(2), 13 (2008)
16. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features–enhancing flexibility in process-aware information systems. Data & knowledge engineering 66(3), 438–466 (2008)