Graph-Based Pattern Identification from Architecture Change Logs

Aakash Ahmad, Pooyan Jamshidi, and Claus Pahl

Lero - The Irish Software Engineering Research Center
School of Computing, Dublin City University, Ireland
{ahmad.aakash,pooyan.jamshidi,claus.pahl}@computing.dcu.ie

Abstract. Service-based architectures have become commonplace, creating the need to address their systematic maintenance and evolution. We investigate architecture change representation, primarily focusing on the identification of change patterns that support the potential reuse of common changes in architecture-centric evolution for service software. We propose to exploit architecture change logs - capturing traces of sequential changes - to identify patterns of change that occur over time. The changes in the log are formalised as a typed attributed graph that allows us to apply frequent sub-graph mining approaches to identify potentially reusable, usage-determined change patterns. We propose to foster the reuse of routine evolution tasks to allow an architect to follow a systematic, reuse-centered approach to architectural change execution.

Keywords: Service-driven Architecture, Change Patterns, Evolution.

1 Introduction

Software architecture represents the global system structure for designing, evolving and reasoning about the configurations of computational components and their interconnections at higher abstraction levels. Service-Oriented Architecture (SOA) is an architectural approach that models business processes as technical software services. Once deployed, continuous change in business and technical requirements leads towards frequent maintenance and evolution in service systems [12]. In order to accommodate recurring changes in the SOA lifecycle, the solution lies in developing processes, frameworks and patterns that enable change reuse for architectural evolution of service software [12].

We have been working on the 'Pat-Evol' project [2, 3] that aims at supporting pattern-driven reuse in architecture-centric evolution for service-driven software. Based on the taxonomy of software change [1], we believe that a systematic investigation of architecture change history could help us to discover sequences of recurring change that occur over time. Recurring changes can be exploited to identify change patterns that support a generic, potentially reusable solution to recurring architecture evolution problems. Therefore, we focus on change representation and its operationalisation by maintaining an architecture change log -

tracing each individual change - for our case studies. The change log keeps a sequential history (as the 'post-mortem' data) of architectural changes, providing us with an empirical foundation to identify patterns of change.

Although a recent emergence of evolution styles [5, 13, 6, 7] promotes the 'build-once use-often' philosophy in architecture and process evolution, it falls short of addressing frequent demand-driven process-centric changes [15, 16] that are central to maintenance and evolution of SOAs. This motivates the needs to systematically investigate architecture change representation that goes beyond frequent addition or removal of individual components and connectors to operationalise recurrent process-based architectural changes.

The proposed solution is based on formalising architectural changes from the logs as a typed attributed graph [8] that provides formal semantics with its node and edge attribution to operationalise architectural changes [9]. We utilise frequent sub-graph mining [4] techniques to not only identify the exact instances, but also inexact matches where only central pattern features suffice for identification. The scalability of solution beyond manual analysis is supported with a prototype 'G-Pride' (Graph-based Pattern Identification) that facilitates automation and parameterised user intervention for pattern identification process. We believe, a continuous experimental identification of patterns is the first step towards facilitating the architect(s) to capitalise on a reuse-centered approach to systematically accommodate recurring changes in existing software.

This paper is organised as follows. A formal specification for the change pattern(s) is presented in Section 2, followed by an overview of the proposed solution in Section 3. We elaborate on graph-based pattern identification in Section 4 and its evaluation in Section 5. In order to justify the overall contribution, related work is presented in Section 6 that is followed by conclusions.

2 Change Pattern

In change logs, we observed that the operationalisation of individual changes represent a parameterised procedural abstraction. This helps us to define change pattern as a *generic, first class abstraction (that can be operationalised and parameterised) to support potentially reusable architectural change execution.* We present a formal description of change pattern in terms of a meta-model of its constituent elements in Figure 1a along with its properties in Figure 1b.

2.1 Pattern-Based Architecture Evolution

We model pattern-based evolution $PatEvol = < SArch, OPR, CNS, PAT >$ as 4-tuple with element inter-relationships in Figure 1a as explained below.

1. **Service Architecture (SArch)** refers to the architecture elements to which a pattern can be applied during change execution. We utilise attributed typed graphs [8] that provide formal syntax and semantics with its node and edge attribution to model typed instances of architectural elements. We use

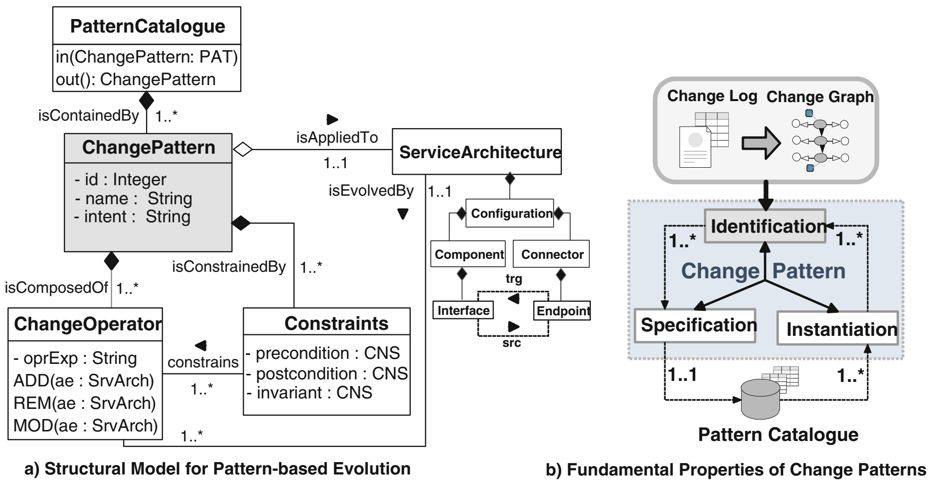

Fig. 1. Model Representation for Reusable Architecture Evolution

the Graph Modeling Language (.GML) for an XML-based representation of architectural instances. The architectural model is consistent with the Service Component Architecture specifications that include configurations (CFG) among a set service components (CMP) as the computational entities that are linked through connectors (CON), in Figure 1a. The modeling is restricted to service-based architectures that only support composition or association type dependencies among service composites. Thus, the structural integrity of architecture elements and consistency of pattern-based change beyond this architecture definition is undefined.

2. **Change Operator (OPR)** represents operationalisation of changes that is fundamental to architectural evolution. Our analysis of the log goes beyond basic change types [1] to define a set of atomic and composite operations enabling structural evolution by adding (ADD), removing (REM) and modifying (MOD) the architecture elements (AE). An inherent benefit of graph-based modeling is the support for architectural evolution by means of graph transformations. More specifically, during change execution the operations could be abstracted as graph transformation rules (in our case supported by XML transformations using XSLT). This enables a fine-granular operationalisation $OPR(ae \in AE)$ to preserve the compositional hierarchy of architecture elements during change execution with:

   - *Atomic Change Operations:* enable fundamental architectural changes in terms of adding, removing or modifying the service operation (OPT), service interface (INF), connector binding (BIN), connector endpoint (EPT) and configuration interface ($cfgINF$).

   - *Composite Change Operations:* are a set of atomic change operations, combined to enable composite architectural changes. These enable adding, removing or modifying the components (CMP), connectors (CON) and configurations (CFG) with a sequential composition of architectural changes.

3. **Constraints (CNS)** refer to a set of pattern-specific constraints in terms of pre-conditions (PRE) and post-conditions ($POST$) to ensure consistency of pattern-based changes. In addition, the invariants (INV) ensure structural integrity of individual architecture elements during change execution.
4. **Change Patterns (PAT)** represents a recurring, constrained composition of change operationalisation on architecture elements that is specified as: $PAT_{<id,\ name>} : PRE(ae_m \in AE) \xrightarrow{INV(OPR_n(ae_m \in AE))} POST(ae'_m \in AE)$. Constraints enforcement on operational composition ensures structural integrity of architecture elements during pattern-based change execution.

A **pattern catalogue (CAT)** refers to a template-based repository infrastructure to facilitate automated storage (in: once-off specification) and retrieval (out: multiple instantiation) of change patterns during evolution.

2.2 Fundamental Properties of Change Pattern

In addition to the meta-model, the fundamental properties of change pattern are presented in Figure 1b. In order to capitalise on pattern-driven reuse, these properties support our argument about change pattern as a generic solution that can be i) identified as recurrent, ii) specified once and iii) instantiated multiple times to support potentially reusable architectural change execution.

- *Identification:* aims at an empirical investigation about the history of architectural changes to identify recurring sequences of change that occur over time. The motivation for architectural change investigation is to discover and analyse real changes (i.e., not any assumed data sets) by extracting the implicit evolution-centric knowledge from change logs, our focus in this paper.
- *Specification:* after identification, it is vital to provide a consistent (once-off) specification for the collection of identified change patterns as pattern catalogue. A template-based specification facilitates flexible querying and retrieval whenever a need for pattern usage arises.
- *Instantiation:* in order to realise the concept of pattern-driven change execution, it allows instantiation of appropriate pattern(s) from its abstract specification to promote the concept of ‘specify-once, instantiate-often’ approach during architecture evolution.

3 Automating Change Pattern Identification

We investigated architectural changes empirically - analysing change representation - to discover recurrent sequences in a change log. Therefore, we have based the identification of patterns on the analysis of changes for two service-based system evolutions we recorded in the log. These include an Electronic Billing Presentment and Payment (EBPP) system whose specifications are published by NACHA and an on-line Tour Reservation System (TRS). We propose a three-phase approach to identify architecture change patterns, in Figure 2.

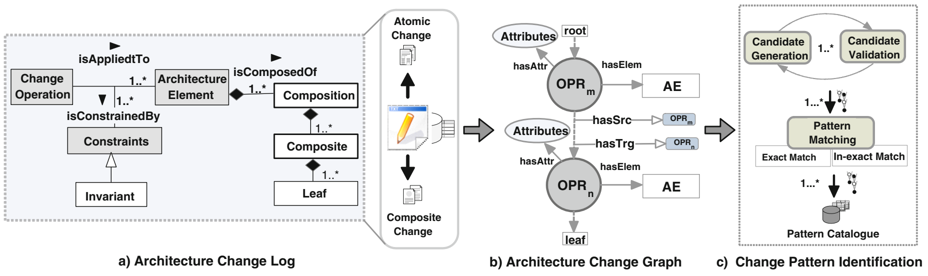

Fig. 2. An Overview of the Proposed Pattern Identification Process

3.1 Maintaining the Architecture Change Log

As the initial step, we follow [1] to record the architecture 'change history'. We use a centrally manageable repository to record sequential architectural changes that are constrained by a set of invariants. We expand on the idea of process change logs from [16] and tailor it to record each individual architectural change as the log tuple. The structural view for the change log is presented in Figure 2a that acts as the foundation to identify change patterns with specific frequency thresholds. While analysing the change operationalisation, each individual change from the case studies is manually recorded in the log that currently comprises of more than a couple of thousands of changes. (i.e., $OPR > 2000$). The primary benefits of this approach included:

- Maintaining the traces of evolution in an updated central repository.
- Analytical support with searching and querying concrete instances of change.
- Experimental analysis of change representation, patterns identification etc.

In Figure 2, the structure of change log maintains the compositional hierarchy of elements. For example, every service component (Composition) must contain at least one or more interfaces (Composite) containing one or more operations (Leaf), while connectors must have binding that contain sn endpoint. We are specifically interested in analysing recurrent sequences that exhibit process-centric changes (e.g. integration, replacement, decomposition etc.) that are central to SOA evolution, as composite changes based on addition or removal of individual components and connectors.

3.2 Graph-Based Formalisation of Architectural Changes

We formalise the architectural changes in the log as an attributed graph (AG) with its nodes and edges typed over an attributed typed graph (ATG) [8] using an attributed graph morphism $t : AG \rightarrow ATG$ as indicated in Figure 3. The ATG provides formal semantics to represent atomic and composite changes with visualisation, efficient searching and analysis of sequential changes in the log. However, we are specifically interested in exploiting frequent sub-graph mining to identify recurring sequences as potential change patterns.

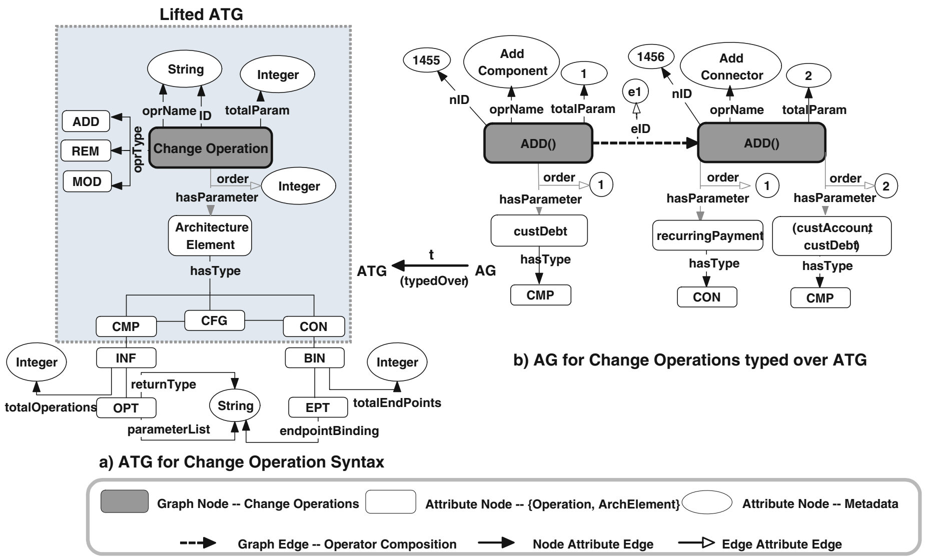

Fig. 3. Attributed Graph to Formalise Architecture Change Operationalisation

Lifting the Change Graph - Sequential Composition: In the change log, analysing an individual change lacks the required abstraction to exploit the recurrent process-centric changes. Furthermore, taking into consideration the granularity of architectural changes (OPR in Section 2) there does not exist a unified representation for architectural evolution that satisfies the needs for all stake-holders' view. For example, a software developer might be more interested in analysing the modification of a specific operation's signature and their semantics, while the architect would be exclusively concerned about the affected component-level interconnections. The possible views could be virtually unlimited depending on any specific evolutionary perspectives. However, in this paper, instead of focusing on atomic changes we focus on sequential composition identification that exhibits process-centric aspects of change in terms of integration, replacement, decomposition of elements. Therefore, we apply graph lifting [11] to enable projection onto higher-level architectural composites that include configurations, components and connectors, hiding low-level atomic changes.

Creating the Change Session Graph: Once the graph is lifted, we provide a utility method as sessionGraph(uID, strTime, endTime) to automatically create the change graph based on a particular change session in the log. The change session is determined by the identification of the user (uID) who applied the change(s) in a specific time interval ($endTime$ - $strTime$). For experimental purposes, we consider all the changes in the log as a single session to extract the attributes of change instances that, we generate the lifted change graph (Figure 3a - dotted blue square) with a concrete represention using the Graph Modeling Language (.GML) format. The result is a directed graph representing sequential composition of change operationalisation, illustrated in Fig. 3b.

For clarity of presentation, some additional attributes (like date, time, committer of change etc.) from the actual graph are hidden to focus on the sequencing of operations on architecture elements. The attributed graph morphism $t : AG \to ATG$ is defined over graph nodes with $t(MetaData) = \mathsf{ChangeData}$ that results in $t(ChangeOperation) = \mathsf{ADD()}$, $t(ArchitectureElement) = $ custDebt, recurringPayment, custAccount and $t(hasType) = $ CMP, CON in Figure 3. The change operationalisation as a typed attributed graph is expressed as 5-tuple: $G_C = \ < N_G, N_A, E_G, E_{NA}, E_{EA} >$, with:

1. **Graph Nodes** $N_G = \{n(g_i) | i = 1, ..., k\}$ is the set of graph nodes. Each node represents a single change operation (i.e., add a component, remove a connector etc.), where t(N_G) = ADD(), REM(), MOD().
2. **Attributed Nodes** $N_A = \{n(a_i) | i = 1, ..., l\}$ is the set of attribute nodes. Attribute nodes are of two types: i) attribute nodes with change metadata, e.g. change operation, name, number of parameters and ii) attribute nodes representing architecture elements (and their compositions) e.g. configuration, component, connector etc, where t(N_A) = (AE : hasType).
3. **Graph Edges** $E_G = \{e(g_i) | i = 1, ..., k - 1\}$ is the set of graph edges which connect source $n(g)_{src}$ and target $n(g)_{trg}$ nodes. It represents the sequencing of change operations in the graph, where t(E_G) = eID($N_{Gi_{src}}$, $N_{Gi_{trg}}$).
4. **Node Attributed Edges** $E_{NA} = \{e(na_i) | i = 1, ..., m\}$ is the set of node attribute edges which join a graph node n(g) to an attribute node n(a), where t(E_{NA}) = nodeAttr(String), e.g. *nID, oprName, totalParam.*
5. **Edge Attributed Edges** $E_{EA} = \{e(ea_i) | i = 1, ..., n\}$ is the set of edge attribute edges which join a node attribute edge e(na) to an attribute node n(a), where t(E_{EA}) = edgeAttr(String), e.g. *eID, eName.*

For example in Fig. 3b, the attributed graph represents two change operations extracted from EBPP architectural changes. It illustrates the addition of a new service component (custDebt hasType CMP) that is connected to an existing component (custAccount hasType CMP) with a connector (recurringPayment hasType CON). The graph nodes are linked to each other using graph edges e(g) having edge id (*e1*) along with the ids of its source and target nodes (*1455, 1456*) representing the applied sequence of change operations.

4 Graph-Based Identification of Change Patterns

Once architectural changes in the log are formalised as an architecture change graph G_C, the final step involves graph-based identification of change patterns. We exploit one of the classical approaches for pattern mining with sub-graph isomorphism [4] from recurring sub-graphs G_P to G_C, where $G_P \subseteq G_C$.

4.1 Properties and Types of Change Sequences

Operationalising the change representation is particularly beneficial to define sequential composition of change operations on architecture elements. This allows

Table 1. Change Sequences (S_x and S_y) as Extracted from the Change Log

	Sequence 1 (S_x)		**Sequence 2 (S_y)**		
cID	**OPR**	**Architecture Elements**	**cID**	**OPR**	**Architecture Elements**
77	REM()	getInvoice $\in$ CMP	312	ADD()	paymentType $\in$ CMP
78	ADD()	custBill $\in$ CMP	313	REM()	custPayment $\in$ CMP
79	REM()	payInvoice $\in$ CON	314	REM()	getBillData $\in$ CON
80	ADD()	payBill $\in$ CON	-	-	-

us to abstract the individual changes into a sequence of recurring change operations representing potential patterns determined by the following properties.

In order to exemplify the properties, Table 1 represents two change sequences (S_x and S_y) extracted from the change log. The sequences contain change id (*cID*), change operation (*OPR*) and the affected architecture element (*AE*). Note, that for space reasons we do not explicitly represent the parameters for connectors as they are insignificant during sequence matching. Sequence 1 (S_x) represents the replacement of the existing component getInvoice with custBill and the corresponding connectors payInvoice, payBill. Sequence 2 (S_y) represents the addition of a new component paymentType that is followed by removal of an existing component custPayment and a connector getBillData.

Type Equivalence (TypeEqu) refers to the equivalence of two change operations given by the utility function $TypeEqu(OPR_1(ae_i : AE), OPR_2(ae_j : AE)) : returns < boolean >$. It depends on the type of change operation and the architecture element for a change operation to categorised as type equivalent (return true) or type distinct (returns false). For example, the change operation $REM(getInvoice \in CMP)$ is only equivalent to $REM(custPayment \in CMP)$ and TypeEqu(77, 313) returns *true*, as in Table 1.

Length Equivalence (LenEqu) refers to the equivalence of length of two change sequences where length of a change sequence is defined by the number of change operation contained in it. It is given by the function $LenEqu(S_x, S_y) : returns < integer >$. Therefore, the length equivalence of two change sequences S_x and Sy is determined by the numerical value (0 imples $S_x == S_y$, -n implies $S_x < S_y$ by n nodes and +n implies $S_x > S_y$ by n nodes). For example, in Table 1 the length of $S_x > S_y$ by one operation so TypeEqu(S_x, S_y) returns 1.

Order Equivalance (OrdEqu) refers to the equivalence in the order of change operations of two sequences. Analysing the change log based on a given change session, we observed that it is normal for same user to perform similar changes using different sequencing of change operations. The semantics and impact of change operation remains the same even if sequencing of change operations is varied. It is given by the function $OrdEqu(S_x, S_y) : returns < boolean >$. We distinguish different types of identified sequences, in Table 2.

- *Exact Sequence:* Two given sequences are exact subsequences if they match on operational types, length equivalence and the ordering of the change operations. In Table 1, S_x and S_y are not the exact sequences because in both the sequence length and the order of operation do not match.

Table 2. The Types of Identified Sequences in the Change Log

Sequence Type	**TypeEqu**	**LenEqu**	**OrdEqu**
Exact Sequence	true	0	true
Inexact Sequence	true	0	false
Partial Exact Sequence	true	$\pm$ n	true
Partial inexact Sequence	true	$\pm$ n	false

- *Inexact Sequence:* Two given sequences are inexact matching sequences if their operational types and lengths are equivalent, but order of change operation varies. In Table 1 S_x and S_y are not the inexact matching sequences as $S_x > S_y$.

- *Partial Exact Sequence:* Two given sequences S_x and S_y are partially exact such that (if $n > 0$ than $S_y \subset S_x$, or if $n < 0$ than $S_x \subset S_y$) - however, the types and ordering of the change operations in the matched sequences must be equivalent. In Table 1 S_x and S_y are not partial exact matching sequences as the order of operations in both the sequences do not match.

- *Partial Inexact Sequence:* Two given sequences S_x and S_y are partial and inexact if (if $n > 0$ than $S_y \subset S_x$, or if $n < 0$ than $S_x \subset S_y$); in addition, the operations within both sequences must be type equivalent, while the order of change operations in both sequences varies. In Table 1 S_x and S_y are partial inexact match. Although $S_x > S_y$, still $S_y \subset S_y$ as cID(77, 78, 79) matches cID(312, 313, 314) ($\mathsf{OrdEqu}(S_x, S_y)$ returns *true*).

4.2 Pattern Identification Process

The properties in Table 2 are vital to not only identify exact instances, but also inexact matches and possible variants of a change pattern where only some pattern features suffice for identification. We introduce the pattern identification problem[1] as a modular solution. This enables automation along with appropriate user intervention and customisation through parameterisation in Table 3 for pattern identification. We follow an apriori-based approach that proceeds in a generate-and-test manner using a Breadth First Search strategy during each iteration to i) generate and ii) validate pattern candidates from G_C and finally, (iii) determine the occurrence frequency of exact and inexact candidates in G_C.

Candidate Generation. The initial step of pattern identification generates a set of candidate sequences S_C from change graph G_C. A candidate consists of a number of nodes representing change operationalisation on architecture elements as a potential pattern depending on its occurrence frequency $Freq(S_C)$ in G_C. *Input(s)* is the change graph G_C and user specified minimum $minLen(S_C)$ and maximum $maxLen(S_C)$ candidate sequence lengths. *Output(s)* is a list of generated candidates $List(S_C)$ such that $minLen(S_C) \leq Len(S_{C_i}) \leq maxLen(S_C)$.

Candidate Validation. We observed that during candidate generation, there may exist some false positives in terms of candidates that violate the struc-

[1] The algorithms along with the developed prototype can be accessed at: `http://www.computing.dcu.ie/~aaakash/ChangePattern.html`

Table 3. List of User Specified Parameters for Pattern Identification

Parameter	**Description**
G_C	Change session graph created from change Log.
S_C	Candidate sequences generated from change graph: $S_C \subseteq G_C$.
G_P	Identified Pattern instance from change graph: $G_P \subseteq G_C$.
$Len(S_C)$	Candidate length representing number of change operations in S_C.
$minLen(S_C)$	Minimum candidate length that is specified by the user: $minLen(S_C) \leq Len(sc) : sc \in S_C$.
$maxLen(S_C)$	Maximum candidate length that is specified by the user: $maxLen(S_C) \geq Len(sc) : sc \in S_C$.
$Freq(S_C)$	Frequency threshold that is specified by the user for S_C to be identified as a pattern G_P.
$List(param : G_C)$	The list of candidates S_C or patterns G_P as $param \subseteq G_C$.

tural integrity of architecture elements when identified and applied as patterns. Therefore, it is vital to eliminate such candidates through validation for each generated candidate sequence *sc* against architectural invariants before pattern matching. *Input(s)* is a candidate $sc \in G_C$ (called from candidateGeneration(), each time a candidate is generated). *Output(s)* a boolean value indicating either valid (true) or invalid (false) candidate sequence *sc*.

Pattern Matching. The last step identifies exact and inexact instances of change patterns based on a user specified frequency threshold. This is achieved by structural matching using sub-graph morphism [4] among the nodes of $List(S_C)$ to corresponding nodes in G_C. *Input(s)* is a list of (validated) candidates $vList(S_C)$, specified frequency threshold $Freq(S_C)$ and G_C. *Output(s)* is a list of identified patterns consisting of pattern instance G_P and its frequency $Freq(G_P)$. A given candidate is an identified pattern (exact or inexact) if its frequency is greater or equal to the user specified threshold: $freq(G_P) \geq Freq(C_P)$.

5 Experimental Analysis and Illustration

The identified pattern types are generally classified as Inclusion, Exclusion and Replacement types depending on the impact of change as addition, removal or modification of elements from existing architecture.

5.1 Identified Pattern Instance - Component Integration

In Table 5, we present an identified instance of the co-related Inclusion pattern that is specified as Integrate (CNS, OPR, AE). Such a declarative specification facilitates the retrieval of appropriate patterns from a catalogue, consisting of the syntactical context that contains pattern pre- and post-conditions (CNS), the applied change operations (OPR) and the affected architecture elements (AE). The co-related Inclusion pattern aims at *integration of mediator services among*

Table 4. Template-based Specification of Change Pattern Instance

Name = Corelated Inclusion, Id = 3, CLS = 1, Intent =“...”, Frequency = 8			
Precondition(s): as in Figure 4a, PostCondition(s): as in Figure 4c			
cID	**OPR**	**Architecture Elements**	**Parameters**
1454	ADD()	CustomerAccount $\in$ CMP	“ ”
1455	ADD()	CustomerDebt $\in$ CMP	“ ”
1456	ADD()	recurringPayment $\in$ CON	CustomerAccount, CustomerDebt $\in$ CMP
1457	ADD()	billAmount $\in$ CON	Biller_CRM, CustomerAccount $\in$ CMP
1458	ADD()	paidAmount $\in$ CON	Biller_CRM, CustomerDebt $\in$ CMP
1459	REM()	customerTariff $\in$ CON	Biller_CRM, CustomerPayment $\in$ CMP
1460	REM()	paymentInvoice $\in$ CON	Biller_CRM, CustomerInvoicing $\in$ CMP
1461	ADD()	makePayment $\in$ CON	CustomerAccount, CustomerPayment $\in$ CMP
1462	ADD()	getReceipt $\in$ CON	CustomerDebt, CustomerInvoicing $\in$ CMP

two or more directly connected service components. The column cID represents the sequences of change as it is captured in the change log and later as individual graph nodes. Figure 4 represents a partial architecture view for the EBPP case study (integrate direct debit to customer accounts and adjust customer debt) captured as a recurring sequence ($Freq(S_C) = 8$) in Table 4.

For example, in Figure 4 the preconditions specify the components (*Biller_CRM, CustomerPayment, CustomerInvoicing*) and connectors (*customerTariff, paymentInvoice*) must exist in the architecture before a pattern can be applied. In addition, the post-conditions specify the addition of new components (*CustomerAccount, CustomerDebt*) and connectors (*makePayment, recurringPayment, getReceipt*) as a result of pattern-driven change execution. The change operations specify the execution aspects in terms of addition or removal of specified elements from the architecture, illustrated in Figure 4b.

5.2 Algorithmic Analysis

Pattern identification from change logs, which can potentially be significant in size, requires an efficient solution. In our trials, we observed that the preprocessing for a significant graph size (i.e, $G_C.size() = OPR \geq 2278$) remains constant with average complexity time = 888.9 ms. However, such pre-processing is fundamental to our approach and the benefit for candidate validation lies in eliminating the potential patterns (false positives) that may violate the structural integrity of an architecture. We customise the input parameters as: $minLen(S_C) = 2, maxLen(S_C) = 9$ with total change operations: $G_C.size() = 2278$. In addition, we increase the pattern frequency threshold $Freq(S_C)$ by 2 in each trial, where $Time \propto Freq(S_C)$ and $Freq(S_C) \propto 1/Instances$. The technical difference between the exact and inexact pattern matching is detailed in Section 4. The summary of comparison (on average): time (exact : inexact) in milliseconds = T(564:1214) ms and identified patterns instances (exact : inexact) = I(21:38), for $G_C.size() = 2278$.

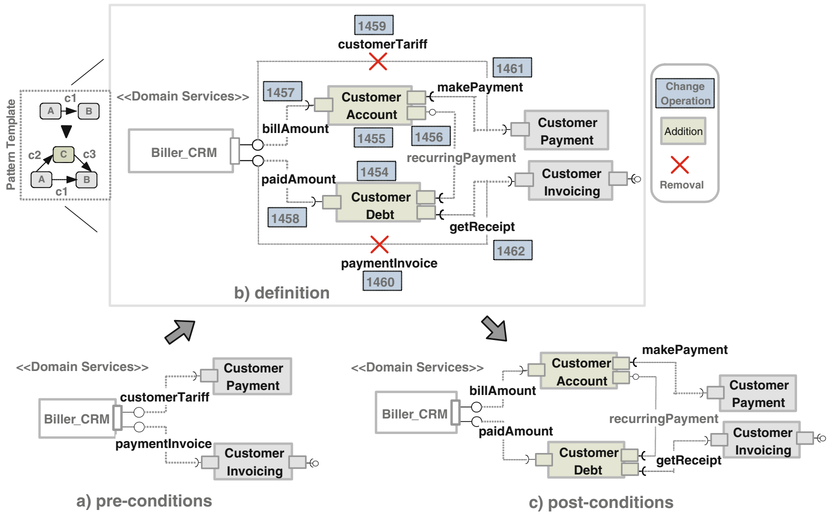

Fig. 4. An Example of the Identified (Co-related Change) Pattern Instance

Possible Limitations: The proposed approach falls short of capturing dynamic dependencies in terms of service compositions that correspond to the behavioral aspects in SOAs. These dynamic dependencies go beyond structural graph matching and is out of the scope for this research. The limitation is inherent in the change log that only captures association type connectors that correspond to structural changes. In addition, change patterns do not necessarily support an optimal solution to architecture evolution problem; instead they promote an alternative and potentially reusable solution.

6 Related Work

Two areas - pattern-driven change reuse and graph-based pattern identification - play a role in our research. The solutions for *pattern-based architecture evolution* utilise evolution styles [5] and more specifically "evolution shelf" [13] as a generic infrastructure to achieve for-reuse and by-reuse techniques for software architecture evolution. It aims at supporting refactoring patterns (i.e., add a component, move a component etc.) that can be composed into further advanced evolution styles (add a client, move a client etc.). In contrast to the evolution styles [5, 13] for more conventional component architectures, we observe that operationalisation of changes in the log exhibits process-centric patterns of change unlike the frequent addition or removal of individual components and connectors.

A catalog of process change patterns [15] can guide change in process-aware information systems. In contrast, we exclusively focus on change operationalisation for architectural abstraction. This allows us to argue about change patterns as generic, first class abstraction that can be specified once and instantiated multiples times to support potential reuse in architecture-based change execution. We follow ideas in [16] that utilise process change logs to gain an empirical insight into the context and scope for process instance changes. Our solution focuses on fostering the common architectural changes that could guide the architects to follow a reuse-centered approach for architectural change execution.

The solution to our pattern identification problem is similar to other *graph-based pattern identification* techniques based on frequent sub-graph mining techniques [4]. We use an apriori-based approach with Breadth First Search strategy for iterative graph matching. In this context, Graph X-Ray (G-Ray) [10] works on attributed graphs to find subgraphs that either match the desirable query pattern exactly, or as close as possible based on pre-defined criteria. We are specifically concerned with identifying patterns in medium to large attributed graphs where graph nodes and edges may have multiple attributes that contain instances of architecture elements and pattern-specific constraints.

7 Conclusions

Service software evolves as a consequence of business and technical change cycles. Scalability beyond manual evolution and change support is required to enable a systematic change reuse during architecture evolution. Investigating the history of sequential architectural changes allows post-mortem analysis to identify patterns as generic, potentially reusable solution for software architecture evolution. The contribution of this paper is a graph-based formalism for architecture change representation that allows automation along with parameterised customisation to identify change patterns.

In the future, we will focus on developing a pattern catalogue as a repository infrastructure to support an automated storage and retrieval of change patterns. We utilise an XML pattern template that allows for once-off abstract specification for identified patterns [17] that can be queried and instantiated with concrete pattern instances to support potentially reusable architecture evolution.

References

1. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a Taxonomy of Software Change. Jrnl of Software Maintenance and Evolution 17, 309–332 (2005)
2. Ahmad, A., Pahl, C.: Pat-Evol: Pattern-drive Reuse in Architecture-based Evolution for Service Software. ERCIM News 88 (January 2012)
3. Ahmad, A., Pahl, C.: Customisable Transformation-Driven Evolution for Service Architectures. In: Europ. Conf. on Software Maintenance and Reengineering, CSMR 2011. Doct. Consort. (2011)
4. Jiang, C., Coenen, F., Zito, M.: A Survey of Frequent Subgraph Mining Algorithms (2004)
5. Garlan, D., Barnes, J., Schmerl, B., Celiku, O.: Evolution Styles: Foundations and Tool Support for Software Architecture Evolution. In: Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture (2009)
6. Gruhn, V., Pahl, C., Wever, M.: Data Model Evolution as Basis of Business Process Management. In: 14th International Conference on Object-Oriented and Entity Relationship Modelling O-O ER 1995. LNCS Series. Springer (1995)

7. Javed, M., Abgaz, Y.M., Pahl, C.: A Pattern-Based Framework of Change Operators for Ontology Evolution. In: Meersman, R., Herrero, P., Dillon, T. (eds.) OTM 2009 Workshops. LNCS, vol. 5872, pp. 544–553. Springer, Heidelberg (2009)
8. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
9. Pahl, C.: A Formal Composition and Interaction Model for a Web Component Platform. In: ICALP 2002 Workshop on Formal Methods and Component Interaction. ENTCS (2002)
10. Tong, H., Faloutsos, C., Gallagher, B., Eliassi-Rad, T.: Fast Best-Effort Pattern Matching in Large Attributed Graphs. In: 13th ACM International Conference on Knowledge Discovery and Data Mining, KDD 2007, pp. 737–746 (2007)
11. Fahmy, H., Holt, R.C.: Using Graph Rewriting to Specify Software Architectural Transformations. In: 15th Intl. Conf. on Automated Software Engineering (2000)
12. Lewis, G., Smith, D.B., Kontogiannis, K.: A Research Agenda for Service-Oriented Architecture (SOA): Maintenance and Evolution of Service-Oriented Systems. Technical report, Software Engineering Institute (2010)
13. Goaer, O.L., Tamzalit, D., Oussalah, M., Seriai, A.D.: Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures. In: 32nd Annual IEEE Intl. Computer Software and Applications Conference (2008)
14. Ng, R., Lakshmanan, L., Han, J., Pang, A.: Exploratory Mining and Pruning Optimizations of Constrained Associations Rules. In: SIGMOD 1998 Conference (1998)
15. Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)
16. Günther, C.W., Rinderle, S., Reichert, M., van der Aalst, W.: Change Mining in Adaptive Process Management Systems. In: Meersman, R., Tari, Z. (eds.) OTM 2006, Part I. LNCS, vol. 4275, pp. 309–326. Springer, Heidelberg (2006)
17. Barrett, R., Patcas, L.M., Murphy, J., Pahl, C.: Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In: International Conference on Web Engineering, ICWE 2006. ACM Press (2006)