Graph-based Pattern Discovery from Software Architecture Change Logs

Aakash Ahmad, Pooyan Jamshidi, Claus Pahl
Lero - the Irish Software Enginneering Research Centre
School of Computing, Dublin City University, Ireland
[ahmad.aakash||pooyan.jamshidi||claus.pahl]@computing.dcu.ie

Abstract: Modern software systems are subject to a continuous evolution under frequently varying requirements and changes in operational environments. Lehman's law of continuing change demands for long-living and continuously evolving software to prolong a system's productive life and economic value with frequent change implementation. We investigate architecture change logs - performing a post-mortem analysis of architectural evolution histories - to discover change patterns that support (a) reusability of architectural changes and (b) enhance the efficiency of the architecture evolution process. We formalise the change log data as a graph and provide the algorithms that utilise sub-graph mining techniques to discover the sequences of recurring change as patterns. The primary contribution of this research is an automated discovery and template-based specification of architecture change patterns from logs. The analysis of change logs have resulted in the discovery of 7 new change patterns and some pattern variants. We documented the patterns and applied them to evolve a peer-to-peer system to a client-server architecture. The proposed algorithms promote pattern discovery as a continuous process and provide a foundation to develop a collection of change patterns that grows overtime with newly discovered patterns.

Key Words: Software Architecture, Software Evolution, Evolution Patterns, Repository Mining.

Category: D.2.10 - Design, D.2.11 - Software Architectures, M.8 - Knowledge Reuse.

1 Introduction

Modern software systems continuously evolve as a consequence of frequent changes in business and technical requirements and their operating environments [1, 2]. Lehman's law of 'continuing change' [3] states that *"... systems must be continually adapted or they become progressively less satisfactory"*. The primary challenges associated to supporting a continuous change are [2, 4] (a) acquisition and application of reusable solutions to address recurring evolution problems and (b) selection of an appropriate abstraction for software change implementation. To address the challenges above, we propose that the acquisition of reusable solutions as discovered change patterns [5] promotes reuse and efficiency in architecture-centric software evolution (ACSE) [6].

Software architecture models proved successful in representing code modules and their interconnections as high-level components and connectors that facilitate the analysis and implementation of software design and evolution at higher abstraction levels [7, 8]. Our systematic reviews of research on ACSE [8, 9] suggests that solutions that tackle recurring evolution problems must rely on a continuous discovery of evolution-centric knowledge that can be reused to guide architecture change management. Some industrial research also demonstrates that reuse knowledge saves up to 40% of the effort for architecture evolution compared to an ad hoc and once-off implementation of recurring architectural changes [10]. In [9] we defined architecture evolution reuse knowledge as *"a collection and integrated representation (problem-solution map) of analytically discovered, generic and repeatable change implementation expertise that can be shared and reused as a solution to frequent (architecture) evolution problems"*.

Evolution styles [7, 11] and change patterns [12] promote the application of reuse knowledge in architecture evolution process. However, there is a lack of research on the acquisition of reuse knowledge that involves a continuous discovery of new styles and patterns. In contrast to the existing research on pattern application [7, 12], there is a need for solutions that support empirical discovery of patterns by investigating pattern sources. In this research we unify the concepts of (a) *software repository mining* [13, 14] and (b) *software evolution* [1, 2] to discover and apply architecture change patterns. First, we apply repository mining techniques on architecture change logs to discover recurring changes as patterns and document them using pattern templates. Second, we utilise software evolution concepts and apply the discovered patterns to support architectural evolution. We hypothesise that:

a continuous experimental investigation of architecture change logs enables the discovery of architecture change patterns that can be shared and reused (to guide the ACSE process).

Research Challenges and Solution Overview **-** based on the hypothesis above, the primary challenges for this research include (a) an automated ***discovery*** of architecture change patterns by mining change logs, (b) a template-based ***specification*** of the discovered patterns, and (c) ***application*** of these patterns to support reuse of architecture evolution. Considering architecture change analysis in [15, 16], in addition to automation; user intervention is also required - human-centric feedback and supervision - for the pattern discovery process. To address these challenges, we provide the solution as a 3-step process that enables the discovery, specification and application of architecture change patterns. In step 1 we capture structural architectural changes - from architecture evolution case studies - in logs and formalise the change log data as a graph. In this research, we only focus on changes that evolve architectural structure, while the analysis of behavioural changes represents a possible dimension of future research. Once log data is formalised as a graph, in step 2 we apply sub-graph mining [17] techniques to identify recurring architectural changes as patterns. Finally, in step 3 a template-based change pattern specification allows us to document individual patterns and enables their reuse whenever the needs for pattern usage arises [18].

Research Contributions **-** a case-study based demonstration highlights the applicability of the discovered patterns to guide architecture change management. In the context of existing solutions for pattern [12] and style-driven [7] evolution, the primary contribution of this research is:

- Exploiting architecture change logs as a source of evolution-centric knowledge that enable post-mortem analysis of architecture evolution histories for automated discovery of reusable change patterns.
- A template-based specification provides a formal documentation for discovered patterns and builds-up the foundation for a collection of architecture change patterns.
- In contrast to the ad hoc and once-off implementation of architectural changes, change patterns increase reusability for frequent change implementation and enhance the efficiency of the architectural evolution process.

This paper provides a significant extension to our previous research [5] and provides: (a) Automation of pattern discovery with algorithms, (b) Documentation of discovered patterns to enhance their reusability and (c) Validation of the discovered patterns with a case study. The scalability of pattern-discovery process beyond manual analysis is supported with a prototype *G-Pride* (Graph-based Pattern Identification) that enables automation and parametrised user intervention for pattern mining from logs.

The remainder of this paper is organised as follows. We discuss the related research in Section 2 and present a meta-model for pattern-based architecture evolution in Section 3. We present the types of change log data and its formalisation as a graph in Section 4. In Section 5, we present the algorithms for change pattern discovery and discuss a template-based pattern specification in Section 6. We evaluate the applicability and the impact of change patterns on architecture evolution in Section 7. Finally, we present the conclusions and future research in Section 8.

2 Related Work

To justify the proposed contribution(s), we provide an overview of the existing - academic and industrial - research on pattern discovery and pattern application. The discussion is guided by our systematic review on pattern-based reuse of architecture evolution [9]. We analyse and compare:

- the state of academic research on pattern discovery (Section 2.1) and pattern application (Section 2.2) in the context of proposed solution, and
- the relevance of the proposed solution to existing industrial studies (Section 2.3) on change reusability in architecture evolution process.

2.1 Change Pattern Discovery

Based on a systematic classification and comparison on existing research in [4, 8, 9], solutions for discovery of architecture change patterns are not well established. More specifically, the only notable work is on the identification of architectural change patterns from object-oriented software [19]. In contrast to the pattern identification by analysing *source codes changes* in [19, 20], our solution discovers patterns by mining the history of *architecture evolution* using change logs. Our solution is also able to discover patterns and pattern variants that is not addressed in [19]. We propose that the discovered patterns could only be reused if they are documented or specified using a pattern template. The novelty of our solution is a 3-step process for pattern-based architecture evolution. We promote patterns as generic and reusable solutions that can be (a) *identified as recurrent*, can be (b) *specified once* and (c) *instantiated multiple times* to support change reuse in ACSE.

Graph-based Pattern Discovery - We discuss the most relevant graph mining approaches to support change pattern discovery - an approach fundamental to our pattern discovery solution. The concept of discovering sequential patterns was first presented in [15]. Since then, there is a growing development of algorithms and mathematical solutions for mining sequential patterns across different domains [21, 22]. Specifically, the solution to our pattern identification problem are Frequent Sub-graph Mining (FSM) techniques [17]. In our solution, we have exploited the concept of sequential pattern mining [15] by using the sub-graph mining techniques [17] to develop pattern discovery algorithms. These algorithms automate the pattern discovery process.

2.2 Change Pattern Application

In contrast to pattern discovery (in Section 2.1), the research on pattern application for architectural evolution is better established. More specifically, in recent years the emergence of change patterns [12] and evolution styles [7, 11] promoted solutions that can leverage reuse knowledge and expertise[1] to tackle recurring problems in architecture evolution. Both the change patterns and evolution styles although conceptually innovative, they build-upon the more conventional philosophy behind design patterns [24] and architectural styles [25] to address evolution-centric issues in software architectures.

Change patterns follow the reuse methods and techniques to offer a generic solution to frequent evolution problems. Pattern-based solutions enable *corrective*, *adaptive*and *perfective* changes (as per ISO/IEC change taxonomy [4]) to support both design-time as well as runtime evolution. In contrast to change patterns, **evolution styles** focus on defining, classifying, representing and

[1] In the *architecture knowledge* [23] community the terms knowledge and expertise represent the empirically discovered solutions that can be shared and reused to support the development and evolution of software architectures [9].

reusing frequent evolution plans. Style-based approaches are limited to addressing the *corrective* and *perfective* changes implemented as design-time evolution and do not support adaptive changes that implement runtime evolution.

In [8, 9], we reported that change patterns and evolution styles promote application of evolution reuse knowledge in ACSE processes. However, there is a clear lack of research on acquisition of evolution reuse knowledge that involves a continuous discovery of new styles and patterns. Therefore, in contrast to existing work on reusable pattern and style application [7, 11, 12], we propose that patterns must be (empirically) discovered by (systematically) investigating pattern sources [26]. To enhance or enable reusable change management there is a growing need for solutions that facilitate a continuous discovery of reuse knowledge as the frameworks and patterns by investigating evolution histories [13, 27] that is the focus of this research.

2.3 Industrial Research on Implications of Reuse on Architecture Evolution

The industrial research can be divided into (a) reusable adaptation plans and (b) survey-based analysis of evolution reuse. In [10], the authors support reuse of adaptation policies to support dynamic adaptation of the architecture for an industrial system called Data Acquisition and Control Service (DCAS). DCAS system is used to monitor and manage highly populated networks of devices in renewable energy production plants. The research demonstrates that reuse of recurring adaptation strategies and policies saves about 40% of the effort for architecture evolution compared to an ad hoc and once-off implementation of adaptive changes [10]. In other research [28], the authors analysed change requests from four different releases of a large telecom system architecture developed by Ericsson over a three-year period. The research highlights that change reuse has resulted in (a) an *increased maintainability* evaluated in cost of implementing architectural change scenarios, (b) *improved testability*, (c) *easier upgrades*, and also (d) *increased performance*. The impact of software reuse, especially exploiting COTS (Commercial Off-The-Shelf) components is essential to enhance reuse of architectural components and their evolution.

Survey-based Study - In an interesting study (The Architect's Mind-set) [29] the authors performed a survey-based analysis in the industry. The authors collected feedback on the importance of architectural knowledge that can be shared and reused to design, develop and evolve software architectures. Based on the results of the feedback, the study reflects the architect's mind-set on architectural knowledge. It concludes that:

... an increase in the efficiency of the architecture evolution process requires increased (initial) effort to integrate reuse knowledge and expertise (empirically discovered and systematically documented solution) in the process. However, the reuse knowledge and expertise has a direct impact on reduced cost and time to implement changes during future software evolution.

Our research can be compared to the work in [10], and we provide an experimental discovery of patterns to support reuse of future changes. However, our research is only focused on design-time evolution of software architectures. The primary focus of our research is to automate pattern discovery and validate pattern reusability.

3 A Meta-model of Pattern-based Architecture Evolution

In software architecture change logs [30], we observed that architectural changes can be operationalised and parametrised to support architecture evolution. More specifically, architecture elements that are added, removed, or modified are specified as parameters of change operations. The recurring architectural changes represent a change pattern as "*a generic, first class abstraction to support potentially reusable architectural change operationalisation*". A typical example of a change pattern is the replacement of a legacy component C1 with a new component C2 represented as Replace (C1, C2). In Figure 1, the meta-model for pattern-based architecture

evolution provides the structural composition of a change pattern and the relationships that exist between pattern elements. For example, the relationship among two of the elements (change pattern and change operators) represents that a change pattern is composed of change operations. The existing architectural description languages [31] do not support an explicit change implementation on architectural model (components and connectors etc.). The proposed meta-model incorporates the architecture model and also provides change operations on this model to evolve architectural descriptions. To enable pattern discovery and pattern-based architectural evolution, we must specify the individual elements of the meta-model from Figure 1 as below.

1 **Specifying the Architecture Model (ARCH) -** the architecture model is composed of the architecture elements to which a pattern can be applied during change execution. We represent the architecture model as topological configurations (*CFG*) based on a set of architectural components (*CMP*) as the computational entities, linked through connectors (*CON*) [7]. Furthermore, architectural components are composed of component ports (*POR*), while connectors are composed of endpoints (*EPT*) to bind component ports. Therefore, consistency of pattern-based change and structural integrity of architecture elements beyond component-based (also service component) architecture model is undefined. We further discuss the architecture model and its evolution in Section 4.

   Change patterns in this paper address component-based software engineering in general and existing research on component-based software architecture and their evolution [7, 11] in particular. We believe that architecture descriptions in a meta-model can be extended to model more conventional object-oriented architectures [19]; however this possibility can only be seen as a future work.

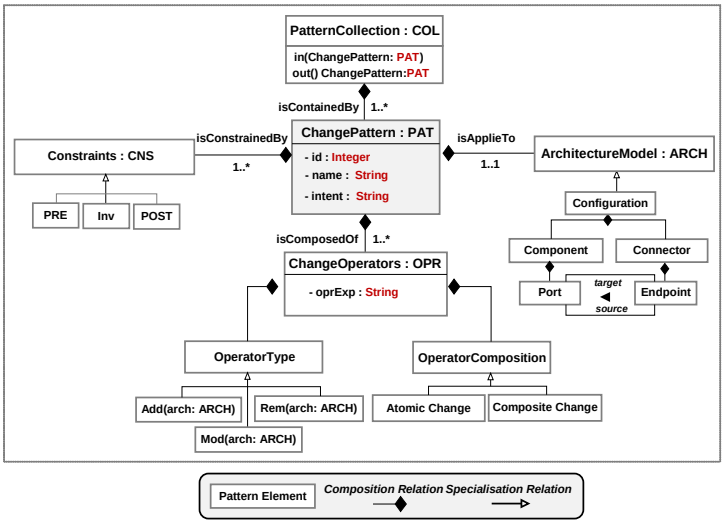


Figure 1: A Meta-model Representation for Pattern-based Architecture Evolution.

2 **Specifying the Change Operators (OPR) -** the change operators represent change instances that are fundamental to operationalising architectural evolution. Our analysis of the change log [30, 32] goes beyond basic change types that address addition (*ADD*), removal (*REM*), and modification (*MOD*) of elements in architecture models [11, 33]. More specifically, first we distinguish between atomic and composite operations and then investigate the

sequential composition of composite operations to discover recurring sequences as change patterns. Architectural composition during change operationalisation is preserved with:

- *Atomic Change Operations:* these enable fundamental changes in terms of adding, removing, or modifying the component ports (*POR*) and connector endpoints (*EPT*). For example, an addition of a new port P in an existing component C is expressed as follows ($\in$ represents type of element).

$$Add(P \in POR, C \in CMP).$$

- *Composite Change Operations:* these are sequential collections of atomic change operations, combined to enable composite architectural changes. These enable adding, removing, or modifying architectural configurations (*CFG*) with components (*CMP*) containing ports, connectors (*CON*) containing endpoints (for component port binding). For example, addition of a new component C with a port P in a configuration G is specified as follows ($\prec$ represents operational sequence).

$$Add(C \in CMP, G \in CFG) \prec Add(P \in POR, C \in CMP)$$

Components are the first class elements (computation and data store) of architecture model. Therefore changes to connectors are consequential, i.e., connectors are only added or removed as a consequence of the addition or removal of components. Change operators represent primitive changes [7] that are composed into pattern-based changes [30]. Operators abstract addition, removal, and modification of components and connectors to support the frequent composition, decomposition, and replacement of architecture elements in the architecture. We further discuss change operations on the architecture model in Section 4.

3 **Specifying the Constraints on Change Operations (CNS) -** the constraints refer to a set of pattern-specific conditions in terms of pre-conditions (*PRE* - the conditions before application of a pattern) and post-conditions (*POST* - the conditions after application of a pattern) to ensure the consistency of pattern-based changes. In addition, the invariants (INV - the conditions satisfied during application of a pattern) ensure structural integrity of individual architecture elements during change execution. For example, during addition of a component C, the preconditions ensure that a component C does not exist in a configuration G, and the postconditions ensure that a component C containing a port P is successfully added to a configuration G. We further discuss the constraints during architecture evolution in Section 4.

4 **Specifying the Change Patterns (PAT) -** a change pattern defines a first-class abstraction that can be operationalised and parametrised to support potentially reusable architectural change execution. A pattern has a *name* and an *intent* that represents a recurring, constrained (CNS) composition of change operationalisation (OPR) on architecture elements ($ae_m \in ARCH$) - in Figure 1. We further discuss pattern discovery in Section 6 and pattern application in Section 7.

5 **A Collection of Architecture Change Patterns (COL) -** the pattern collection is a repository infrastructure that facilitates an automated *storage* (in: once-off specification) and *retrieval* (out: multiple instantiation) of discovered change patterns. It also supports pattern classification for a logical grouping of related patterns based on the types of architectural changes they support. Pattern specification is detailed in Section 6.

The background details about pattern discovery and representation enable us to present the change log data (in Section 4) and pattern discovery from logs (in Section 5).

4 Graph-based Modelling of the Architecture Change Log Data

To exploit the sub-graph mining approaches for frequent pattern mining from logs, we model the log data as a graph. In this section we explain (a) what are the *source* and *types* of change log data (in Section 4.1) and (b) how log data is *formalised as a graph* (in Section 4.2).

4.1 The Source and Types of Change Log Data

In the context of software repository mining research [13], an architecture change log refers to '*an explicit source of evolution-centric knowledge that maintains and provides a sequential collection of architecture change history that has been aggregating over time*' [27, 33]. We define a change log as follows:

Definition 1. Architecture Change Log - Let OPR represent an individual change operation, an architecture change log (ACL) is a sequential collection of change operations expressed as a tuple $ACL =< OPR_1 \prec OPR_2 \prec OPR_N >$.

$\prec$ represents a sequencing operation between change operations (OPR_1 to OPR_N). The change operations represent a sequential collection of architectural changes (Add, Remove, and Modify) on elements of architecture model (components, connectors, and configurations) (cf. Figure 1).

4.1.1 Evolution Case Studies as the Source of Change Log Data

The source of the change log data refers to the architectural evolution case studies [34,35] with all individual architectural changes captured in the log. In this research, we assume that the change logs evolve over time with acquisition of new evolution-centric data from different sources.

Capturing Architectural Changes in the Log is an automated process that is enabled by preserving individual changes in the log as illustrated in Figure 2. This means, whenever an individual change was applied to the architecture elements it was captured in the log. To complement a continuous pattern discovery the algorithms can automate log mining whenever new data is available. Currently, we analyse architectural evolution cases of an a) Electronic Bill Presentment and Payment System (EBPP) [34] and b) 3-in-1 Phone system [35]. To enable experimental investigation of change history, architectural evolution of two different systems provides us with an adequate amount of data for pattern discovery[2] - for space reason only the EBPP case study is used as running example in this paper. The adequacy of the log data is defined as: (a) *granularity of the architectural change instances* (i.e., atomic and composite changes, cf. Section 3), and (b) *total number of changes* for pattern discovery (thousands of individual changes). We present a partial architectural view for the EBPP case study in Figure 2a and explain an evolution scenario to capture architectural change instances[3] in the logs - Figure 2b. We utilise the Architecture Level Modifiability Analysis (ALMA) [37] for evolution scenario elicitation and analysis of EBPP architecture evolution. We follow the ALMA methodology with a three step process for *selection*, *evaluation* and *interpretation* of the evolution scenario.

[2] Each individual architectural change from the case studies is captured in the log file as the basis for pattern discovery from change logs provided here: `http://ahmadaakash.wix.com/aakash#!changelogdata/c22ju`

[3] In literature the terms **architecture change instance** and **architecture change operation** are often used interchangeably [30, 36]. For example a change instance that adds a component C can be operationally expressed as: Add(C $\in$ CMP). Operationalising an instance explicitly provides a name (Add) and parameters (architecture element $\in$ hasType) for a change instance.

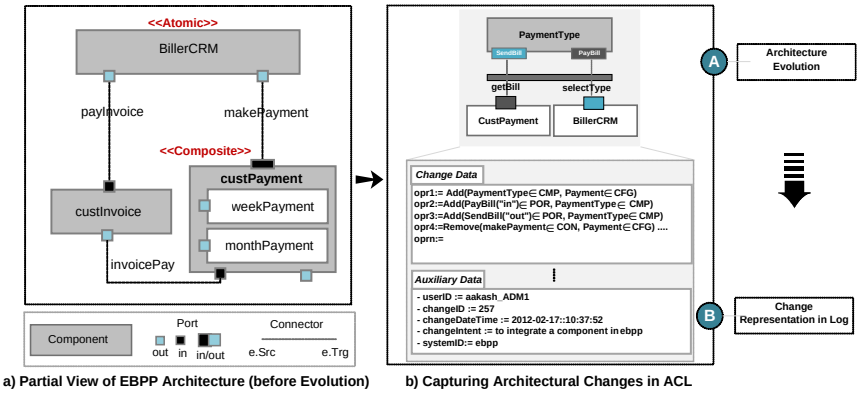


Figure 2: a) Partial Architectural View for EBPP and b) Change Instances in ACL

Step 1. *Scenario Selection - Integration of Architectural Component*: As the first step, scenario selection aims at selecting all (or a subset of) architectural change scenarios for scenario-based analysis of architecture evolution. As an example, we present the evolution scenario of component integration in the EBPP case study. The scenario demonstrates that in the existing functional scope of the case study (Figure 2a), the company charges its customer with full payment of customer bills prior to delivering the requested services. Now, the company plans to facilitate existing customers with either direct debit or the credit-based payments of their bills. In Figure 2b, this evolution scenario is represented as: *integration of a mediator component PaymentType that facilitates the selection of a payment type (direct debit, credit payment) mechanism among the directly connected components BillerCRM and CustPayment*.

Step 2. *Scenario Evaluation - Analysing Architectural Changes for Component Integration*: After the scenario was selected, in this step we are interested in analysing the architectural change operations applied to architecture elements to evaluate and execute the scenario. For example, in the case of component integration, existing EBPP architecture is modified with addition of new components PaymentType (and corresponding ports) and two connector getBill and selectType to mediate the customer billing and payments in Figure 2b. This results in recording individual change operations in the log (change data) along with the intent, time and effects of change (auxiliary data) in Figure 2b.

Step 3. *Results Interpretation - Impacts of Changes on Architecture Model*: After evaluation and execution, as the final step we interpret the results of a given evolution scenario based on the impact of changes on existing architecture. The results interpretation is based on analysing source architecture (as preconditions of evolution) Figure 2a, the change operations applied on source architecture to achieve the evolved architecture (as post-conditions of evolution) Figure 2b.

4.1.2 Types of Data in a Change Log

Once change instances are recorded in the log, change log data is classified as *Change Data (CD)* and *Auxiliary Data (AD)* as represented in Figure 2b.

- *Change Data (CD)*: contains the core information about individual change instances or operations in the log. This is expressed as $CD = (ChangeID, OPR, ARCH)$ representing

change $id(opr_1, opr_2, \ldots, opr_n)$ along with change operations on architecture elements. For example in Figure 2b, change data represents the change id as opr_1 to add a new component PaymentType inside the Payment configuration.

- *Auxiliary Data (AD)*: provides the additional details about individual change instances in the log, representing the time, user, intent of changes. The auxiliary data is expressed as: $AD = (UserID, TimeStamp, ChangeIntent, SytemID)$ that is captured automatically and consists of user id (Aakash-ADM1), date-time (10:37:52/17/02/2012), intent of change (to integrate a component in ebpp) and the system identifier (ebpp) to which the change is performed in Figure 2b. Auxiliary data is particularly useful for architectural change analysis based on the source, intent, time of change and facilitates in extracting specific (time/user-based etc.) architecture change sessions from logs.

4.2 Creating the Change Log Graph

In this section we focus on formalising change instances from log as an attributed graph (AG) with nodes and edges typed over an attributed typed graph (ATG) [38]. Please note that an ATG in Figure 3 represents a meta-graph to model change log data as an AG that represents an instance-graph in Figure 4. An inherent benefit with graph-based modelling of log data lies with exploitation of sub-graph mining - a formalised graph mining technique - whereas recurring sub-graphs in the log graph can be discovered as frequent change patterns.

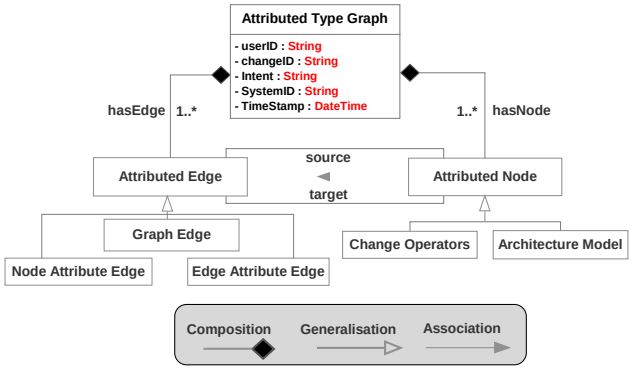


Figure 3: Attributed Typed Graph Model to Formalise Architecture Change Log Data

Definition 2. Architecture Change Log Graph - A collection of change operations (cf. Figure 1) from log (cf. Definition 1) are expressed as a change log graph $G_{ACL} = \langle N_G, N_A, E_G, E_{N_A}, E_{E_A} \rangle$

- **Graph Nodes** represent change operations on architecture model: $N_G, N_A \in Nodes$,
- **Graph Edges** represents a sequencing of the operations as adjacent nodes: $E_G, E_{N_A}, E_{E_A} \in Edges$.

The attributed graph morphism M from an instance graph AG (Figure 3) to its meta-graph ATG (Figure 4) is expressed as $M : AG \rightarrow ATG$. A collection of change operations in the log are expressed as an attributed change log graph G_{ACL} in Figure 3 - nodes and edges defined as:

1. **Graph Nodes:** $N_G = \langle n_g^i | i = 1, \ldots, m \rangle$ represents a set of graph nodes. Each graph node ($n_g \in N_G$) represents a single change log entry (i.e., a single change operation). The sequence $i = 1, \ldots, m$ refers to the total number of change operations that exist in the log. We assume concurrent or commutative change operations (if any in the log) are represented as a sequence, where each of the change operations is executed one after the other (i.e. sequenced change log) [5, 30].

2. **Attribute Nodes:** $N_A = \langle n_a^i | i = 1, \ldots, m \rangle$ represents a set of attribute nodes for graph nodes (N_G). Attribute nodes are of two types, a) attribute nodes that represent auxiliary data (e.g., userID, changeID, TimeStamp etc.) and b) attribute nodes that represent change data and its subtypes (e.g., operation type, architecture model). The sequence $j = 1, \ldots, m$ refers to the total number of attribute nodes in a change log graph.

3. **Graph Edges:** $E_G = \langle n_g^i | i = 1, \ldots, m-1 \rangle$ represents a set of graph edges that connects two graph nodes N_G. The graph edges ($e_g \in E_G$) represent the applied sequence of change operations (OPR) applied to the architecture model (ARCH). The sequence $i = 1, \ldots, m-1$ represents total graph edges in a log graph.

4. **Node Attribute Edges:** $E_{N_A} = \langle e_{na}^i | i = 1, \ldots, p \rangle$ represents the set of node attribute edges that join an attribute node ($n_a \in N_A$) to a graph node ($n_g \in N_G$). The sequence $i = 1, \ldots, p$ refers to the total number of node attribute edges in change log graph.

5. **Edge Attribute Edges:** $E_{E_A} = \langle e_{ea}^i | i = 1, \ldots, q \rangle$ is the set of edge attribute edges that join an attribute node ($n_a \in N_A$) to an attributed edge (e_{na}). The sequence $i = 1, \ldots, q$ refers to the total number of edge attribute edges in a change graph.

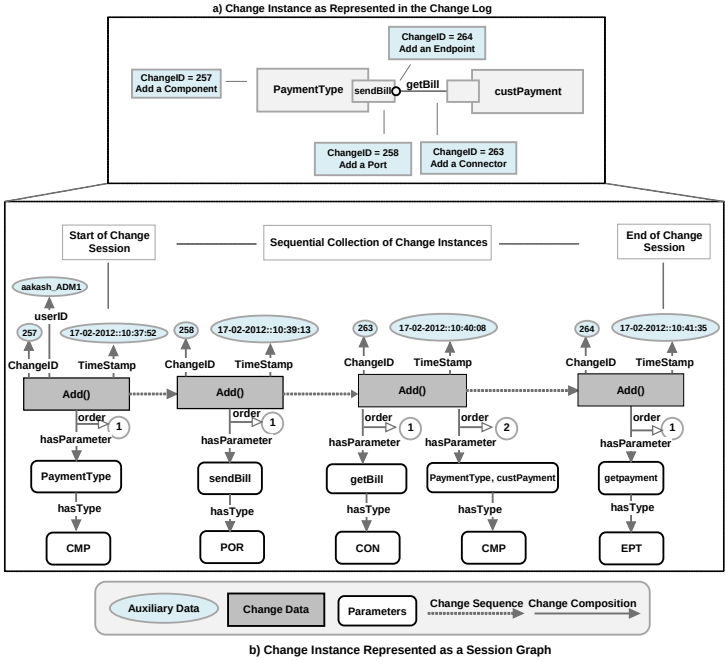


Figure 4: Change Instances as an Attributed Graph (typed over ATG in Figure 3).

4.3 Mapping the Change Log Data to Change Log Graph

After presenting the change log data (Section 4.1) and the change log graph (Section 4.2), we now map the log data (change operations) to the change graph (nodes and edges). Modelling change log data as a graph in Figure 4 allows us exploit the sub-graph mining techniques [17] for an automated discovery of (sequential) change patterns [15]. Continuing with the earlier example (addition of a PaymentType component, cf. Figure 2), in Figure 4 we present a partial view of a log graph that is an instance of a change graph in Figure 3.

In Figure 4, the attributed graph morphism $t : AG \rightarrow ATG$ is defined. This means that the generic elements of ATG (cf. Figure 3) are instantiated with concrete elements of AG in Figure 4. For example, the graph node from $t(ATG) = AG$ is instantiated as t(ChangeOperation) = Add(), t(ArchitectureElement) = PaymentType, custPayment sendBill, getBill, getPayment and t(hasType) = CMP, CON, POR, EPT where (PaymentType, custPayment) hasType CMP, (sendBill) hasType POR, (getBill) hasType CON, (getPayment) hasType EPT. The graph nodes are linked to each other using graph edges e_g for source and target nodes (257, 258, 263, 264) representing the applied sequence of change operations.

The log graph from Figure 4 is represented using the Graph Modelling Language (GML) with additional details in [39]. The GML provides a notation that is manipulated by tools and their underlying algorithms. The GML format provides an XML-based syntax to manipulate the log graph in an automated way. Also, the attributed graph in GML format is an input to the pattern discovery process in Section 5.

5 Graph-based Discovery of Architecture Change Patterns

Once change log data is formalised as an attributed graph [38], the solution to the pattern discovery problem is the application of sub-graph mining approaches [5, 21] on change log graphs. More specifically, our solution to graph-based pattern discovery is mining recurrent sequences (cf. Definition 1) of change operations that is equivalent to discovering sub-graphs which occur frequently in a change log graphs[4] G_{ACL}. In this section, we introduce the pattern discovery problem as a modular solution that enables the parametrisation and customisation of the pattern discovery process.

In Table 1, we provide a list of variables that facilitate the parametrisation of algorithms for pattern discovery. In Table 2, we outline a number of utility functions that are frequently used to maintain the modularity of the pattern discovery process. In order to discover architecture change patterns from logs, we follow a 3-step process illustrated in Figure 5. It consists of the (a) pattern candidate generation, (b) pattern candidate validation and finally (c) pattern matching as detailed in the remainder of this section.

5.1 Algorithm I - Candidate Generation

As the initial step of the pattern discovery process, candidate generation aims at generating a set of pattern candidates P_C from an architecture change graph G_{ACL}, as illustrated in Figure 5a. Each of the generated pattern candidate $p_{c_i} \in P_C$ represents a sub-graph of G_{ACL} as $P_C \subseteq G_{ACL}$. As presented in Table 1, the difference between a pattern candidate and a pattern is that the candidate must satisfy a specific occurrence frequency to be identified as a pattern. Therefore, a pattern candidate represents a change sequence (collection of graph nodes as change operations) as a potential pattern depending on its frequency $Freq(P_C)$ in G_{ACL}. We apply a graph clustering

[4] Please note that the terminology *Change Log Graph*, *Change Graph* or *Log Graph* are used interchangeably that refer to a graph created from change log and is represented as G_{ACL}.

Parameter	Description
G_{ACL}	Architecture change graph created from change Log.
P_C	Pattern Candidate sequences generated from change graph: $P_C \subseteq G_{ACL}$
PAT	Discovered Pattern from change graph: $PAT \subseteq G_{ACL}$
Len(P_C)	Candidate length - number of change operations in pattern candidate P_C
Len(PAT)	Pattern length - number of change operations in change pattern PAT
minLen(P_C)	Minimum candidate length by user: minLen(P_C) $\leq Len(pc)$:pc $\in P_C$
maxLen(P_C)	Maximum candidate length by user: $Len(pc) \geq maxLen(P_C) : pc \in PC$
Freq(P_C)	Frequency threshold by user for P_C to be identified as a pattern PAT.
List($param \in G_{ACL}$)	The list of candidates P_C or patterns PAT $param \subseteq G_{ACL}$

Table 1: Parameters for Graph-based Pattern Discovery process.

Function(param)	Return	Description
$G_{ACL}.size()$	Integer	Get total number of nodes in log graph G_{ACL}
lookUp(P_C)	Boolean	Candidate P_C validation look-up in the invariant table
nodeMatching($n_j; n_k$)	Boolean	Bijective node matching based on $TypeEquv()$ (Section 5.1)
exactMatch($n_i; n_j$)	Boolean	Determine Exact match from candidate P_C to graph G_{ACL}
inexactMatch($n_i; n_j$)	Boolean	Determine Inexact match from candidate P_C graph G_{ACL}

Table 2: A List of Utility Methods for Pattern Discovery.

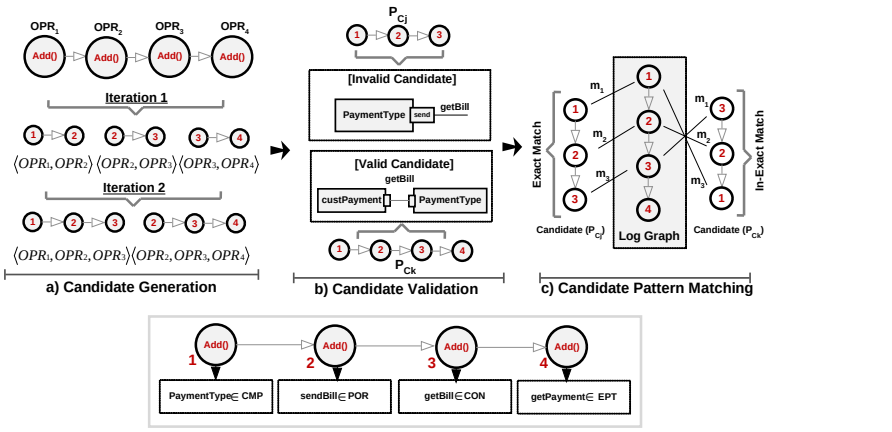


Figure 5: Overview of 3-Step Graph-based Pattern Discovery Process.

approach [40] on G_{ACL} to create graph clusters representing sub-graphs as pattern candidates in Figure 5a. Graph clusters from G_{ACL} are created based on the minimum and maximum length specified by the user as $minLen(P_C) \leq Len(P_C) \leq maxLen(P_C)$ as in Table 1. The size $Len(P_C)$ of a cluster (P_C) represents the total number of nodes in a cluster that ultimately represents the number of change operations in P_C. For example, in Figure 5a the user specifies $minLen(P_C)$: 2 and $maxLen(P_C)$: 3. In the first iteration candidates are generated such that the length of each candidate is two nodes, and with the next iteration each candidate having three nodes. The generation of pattern candidates $PC_1, \ldots, PC_N$ (each representing an individual

pattern candidate (P_C)) based on graph clustering [40] is expressed: *Pattern Candidates* =

$$\left\{ \begin{array}{c} P_{C1} = \langle (OPR_1, OPR_2), (OPR_2, OPR_3), (OPR_3, OPR_4) \rangle \\ P_{C2} = \langle (OPR_1, OPR_2, OPR_3), (OPR_2, OPR_3, OPR_4) \rangle \\ P_{CN} = \langle (OPR_j, OPR_k, \ldots, OPR_n), (OPR_{j+1}, OPR_{k+1}, \ldots, OPR_{n+1}) \rangle \end{array} \right\} \quad (1)$$

1. **Input**: is a user specified change graph G_{ACL} with minimum $minLen(P_C)$ and maximum $maxLen(P_C)$ candidate lengths $minLen(P_C)$: 2 and $maxLen(P_C)$: 3 in Figure 5a.

2. **Process**: starts at the graph root with the selection of a single node and enumerating the temporary candidate list with adjacent node concatenation. Based on $minLen(P_C)$ and $maxLen(P_C)$, a temporary candidate list $buff(P_C)$ is generated as follows: $buff(P_C) = \langle pc_1(OPR_1, OPR_2), pc_2(OPR_2, OPR_3), \ldots, pc_5(OPR_2, OPR_3, OPR_4) \rangle$ (Line 1 - 13). To avoid this exhaustive candidate list, the candidates in $buff(P_C)$ are iteratively matched to find specific candidates that occur at least more than once in G_{ACL}. We use the Breadth First Search (BFS) [21] over G_{ACL} with $nodeMatching(n_i; n_j)$ (Table 2) : $n_i.OPR \xrightarrow{match} n_j.OPR \wedge n_i.ARCH \xrightarrow{match} n_j.ARCH$ to generate the final candidate list $List(P_C)$ (Line 10 - 16). In addition, we ensure each candidate $pc_i \in List(P_C)$ is validated through candidateValidation($cp : G_{ACL}$) (Line 13, cf. Section 5.2).

Algorithm 1: : candidateGeneration()

Input: G_{ACL}, $minLen(P_C)$, $maxLen(P_C)$
Output: $List(P_C)$

```
1:  buff(P_C) ← φ {buffer to hold temporary candidates}
2:  root ← G_ACL.getRoot()
3:  for candLength ← minLen(P_C) to maxLen(P_C) do
4:     maxCandidates ← G_ACL.size() - candLength
5:  end for {get total number of candidates}
6:  while root ≤ maxCandidates do
7:     buff(P_C)_node ← G_ACL(node + root)
8:     candLength ← candLength + 1 {get candidates for validation}
9:  end while
10: List(P_C) ← φ {List to hold validated candidates}
11: for tempCand ← 0 to tempCand ≤ buff(P_C).Length() do
12:    if buff(P_C)_tempCand.Length() == buff(P_C)_Cand.Length() then
13:       if nodeMatching(tempCand, cand) == true and candidateValidation(cand)
          == true then
             List(P_C)_tempCand ← buff(P_C)_cand
14:       end if
15:    end if
16: end for
17: return(List(C_P)) {return list of validated candidates}
```

3. **Output**: is a list of generated candidates $List(P_C)$ such that $minLen(P_C) \leq Len(P_C) \leq maxLen(P_C)$.

5.2 Algorithm II - Candidate Validation

During candidate generation, there may exist some false positives in terms of candidates that violate the structural integrity (invariants) of the architecture model when identified and applied as patterns. For example, in Figure 5 b the candidate P_{C_j} represents three change operations as: *Operation 1* adds a component PaymentType, *Operation 2* adds a port sendBill to component PaymentType, and finally *Operation 3* adds a connector getBill. However, the connector does not provide interconnection with source and target ports (an orphan connector). Therefore, it is vital to eliminate a candidate pattern P_{C_j} that violates architectural integrity (cf. 5b, invalid candidate). In contrast, the candidate P_{C_k} represents four change operations and provides interconnection among component ports in Figure 5b is referred to as a valid candidate. We eliminate invalid candidates through validation for each generated candidate pc against architectural invariants before pattern matching:

1. **Input**: is a candidate $c_{p_i} \in P_C$, $P_C \subseteq G_{ACL}$ (from candidateGeneration() - Line 13).
2. **Process**: includes look-up into the invariant table in terms of validating the configuration of architecture elements in the generated pattern candidates (in Line 3). More specifically it aims at detecting any orphaned components and connectors as a result of associated change operations. The orphaned component has no associated interconnection and orphaned connectors have no associated components, indicated by Boolean value false.
3. **Output**: is a Boolean value indicating either valid (true) or invalid (false) candidate cp.

Algorithm 2: : candidateValidation()

Input: $cp \in G_{ACL}$
Output: $boolean[TRUE/FALSE]$ indicating if a candidate is valid of invalid.

```
1:  isValid ← false
2:  iteration:
3:  for node ← 0 to node ≤ pc.Length do
4:      if lookUp(pc.node.ARCH) == true then
5:          isValid ← true
6:      end if
7:      if isValid ← false then
8:          isValid ← true
9:          break iteration
10:     end if
11: end for
12: return(isValid)
```

5.3 Algorithm III - Candidate Pattern Matching

After candidate validation, the last step involves candidate pattern matching constrained by a user-specified frequency threshold $Freq(P_C)$ for P_C in G_C. If a validated candidate in $List(P_C)$

occurs N times (determined by $Freq(P_C)$), a pattern PAT is discovered in change graph G_{ACL}. We exploit sub-graph isomorphisms to match graph nodes (change operations) of P_C and G_{ACL} iteratively.

1. **Input**: is a list of (validated) candidates $List(P_C)$, specified frequency threshold $Freq(C_P)$ and G_C.

Algorithm 3: : patternMatch()

Input: $List(P_C)$, $Freq(P_C)$, G_{ACL}
Output: $pList(PAT, Freq(PAT))$

```
1: gCand(pc : G_ACL) ← φ {hold extracted nodes from G_ACL}
2: root ← G_ACL.getRoot()
3: for cand ← 0 to cand ≤ List(P_C). Length do
4:    freq ← 0 {to count frequency of P_C in G_ACL}
5: end for
6: while root ≤ G_ACL.getLeaf() do
7:    exactMatch ← 0
8:    inexactMatch ← 0 {set exact, inexact match to zero}
9: end while
10: if List(P_C)_cand.Length() == gCand(root).Length() then
11:    if match(List(P_C)_cand.node, gCand(root).node == true) then
12:       exactMatch ← exactMatch + 1 {exact match found}
13:    end if
14:    if inexactmatch(List(P_C)_cand.node, gCand(root).node == true) then
15:       inexactMatch ← inexactMatch + 1 {inexact match found}
16:    end if
17: end if
18: if exactMatch == List(P_C)_cand.Length() OR inexactMatch ==
    List(P_C)_cand.Length() then
19:    freq++ {increment frequency of pattern discovered}
20: end if
21: if freq ≥ Freq(P_C) then
22:    pList(PAT, Freq(PAT)) ← (List(P_C)_cand, freq)
23: end if
```

2. **Process**: includes retrieving each candidate from $List(P_C)$ and finds its exact or possible inexact instance in G_{ACL}. In a match from P_C to G_{ACL} the number of nodes must be equal (Line 10). We exploit the change operation properties (cf. Definition 1) to specify: if and only if all the nodes in the candidate match the corresponding nodes in a change graph, then P_C is isomorphic to G_{ACL} as: $nodeMatching(P_C, G_{ACL})$ =

$$\begin{Bmatrix} \langle P_{C_1}(OPR_1, OPR_2)\rangle & \cdots & \langle P_{C_n}(OPR_i, OPR_j, \ldots, OPR_k)\rangle \\ \vdots & \ddots & \vdots \\ G_{ACL}(OPR_1, OPR_2, \ldots, OPR_N) & & G_{ACL}(OPR_1, OPR_2, \ldots, OPR_N) \end{Bmatrix} \quad (2)$$

- *Exact Match* : It is based on exact sequences in Table 2 (cf. Section 5.1). An exact match requires that there must exist a bijective mapping among types of change operator and the type of architecture element in attributed nodes that is given as a utility function (cf. Table 2) exactMatch($nodeMatching(n_i; n_j)$)$[\forall(i, j) = 1 \dots N]$ that utilises the function (Table 2) $nodeMatching(n_i, n_j)$ method it enables finding an exact match among the candidate nodes P_C (node) to the corresponding nodes in the change graph G_{ACL} (node) in Figure 5 c. In addition, the ordering of matching nodes from $List(P_C)$ to G_{ACL} must be same. If such an exact instance is found, the candidate's frequency is incremented and matching is repeated (Line 11, 12).

- *Inexact Match* : It is based on in-exact sequences in Table 2 (cf. Section 5.1). The order of matching nodes from $List(P_C)$ to G_{ACL} is not always the same. Foe example, $inexactMatch(nodeMapping(n_i; n_j))[\forall(i \rightarrow j) = 1 \dots N]$ utilises $nodeMatching(n_i, n_j)$ to find an inexact match among the candidate nodes P_C (node) to the corresponding nodes in the change graph G_{ACL}(node) in Figure 5 c. The candidate's frequency is incremented and matching is repeated until leaf node (Line 14, 15).

3. **Output**: is a list of identified patterns consisting of the pattern instance PAT and its corresponding frequency $Freq(P_C)$. A given candidate is an identified pattern (exact or inexact) if its frequency is greater or equal to a user-specified frequency threshold: $freq(PAT) \geq Freq(P_C)$.

5.4 Overview of Prototype for Pattern Discovery

After the discussion of the algorithms for pattern discovery, we now present the individual elements of the user interface of the prototype to highlight process automation and parametrised customisation as in Figure 6.

A ***Log File Selection*** as presented in Figure 6, the prototype allows a user to select a specific change log graph file to start the pattern discovery process. Details about change log graph are already presented in Section 4.2.

B ***Pattern Discovery Parameters*** facilitate a user of the prototype to customise the pattern discovery process. We have provided the details of parameters for pattern discovery in Table 1 as they allow a user to specify:

- *Minimum and Maximum Length of the Pattern Candidate:* as discussed in Section 5.1, a precondition to pattern discovery is the generation of pattern candidates. Therefore, specifying the minimum and maximum length of the pattern candidates allows a user to specify the exact minimum (3 change operations) and exact maximum (10 change operations) length of pattern candidates in Figure 6.
- *Pattern Frequency Threshold:* as in Section 5.1, the user can also specify the pattern frequency threshold. It maintains a minimum frequency (3 occurrences) that must be satisfied to consider the recurring candidates as a discovered pattern.
- *Discovery of Exact and Inexact Pattern Instances:* Section 5.2 distinguished between exact and in-exact pattern instances. The prototype allows a user to specify if they want to discover both exact (23 patterns) as well as inexact (9 pattern) instances. If the user only specifies Exact Pattern Instances, the pattern discovery process is considerable faster but it skips the inexact pattern instances.

C ***Pattern Discovery Results*** as presented in Figure 6 provides a summary of the results for pattern discovery process. It highlights the total number of change operations investigated for pattern discovery. The number of exact as well as inexact patterns instances discovered and the total time taken for pattern discovery.

The discovered patterns need to be specified in a change pattern template. We discuss the prototype support for change pattern specification in Section 6.

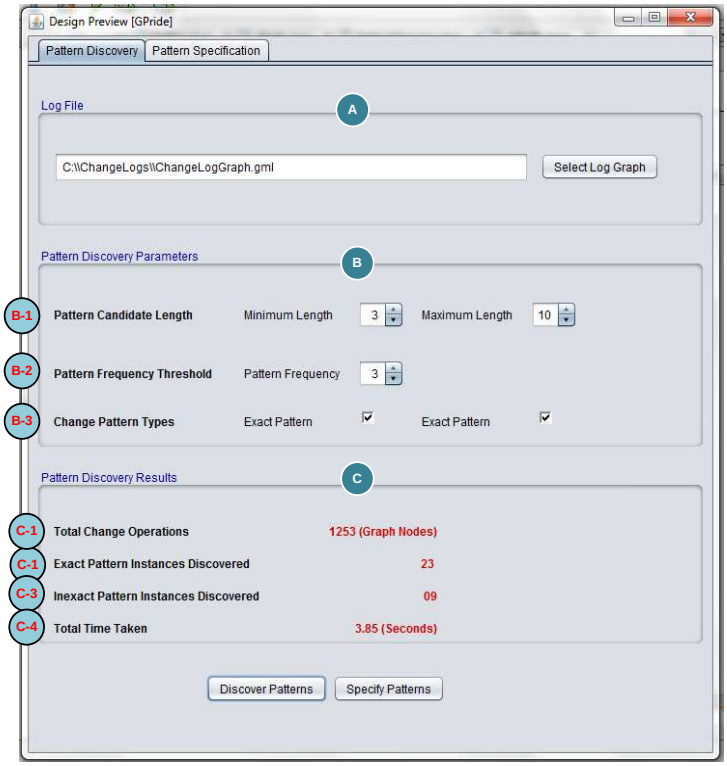


Figure 6: Screen-shot of Prototype for Change Pattern Specification.

6 A Template-based Specification of Discovered Change Patterns

In this section, we present a template-based specification of the discovered change patterns that facilitate pattern reuse. A template provides a structured format to document the individual patterns in terms of pattern *name*, the *intent* of the pattern, its *impact* on the architecture model and other related information. In the remainder of this section, we explain the pattern specification process (in Section 6.1) and provide an overview of the solution (in Section 6.2).

6.1 Specification of the Architecture Change Patterns

We provide a formal template for pattern specification that is based on the meta-model for pattern-based evolution (Section 3) and the guidelines for documenting patterns and styles presented in [18, 41]. We have provided a prototype presented in Figure 7 that allows a user to specify the change patterns in a change pattern template. We exemplify the specification for one of the discovered patterns named the *Component Mediation* pattern - an overview of all the discovered (and specified) patterns is provided later in Section 6.2. Here we focus on the role of prototype to facilitate a pattern author to document the patterns in a semi-automated way. The prototype visualises the impact of a change pattern on the architectural model as well as the constraints

and change operators. Based on pattern overview, the user provides the name and intent of the given pattern to complete a template-based pattern specification following a 3-step process given below.

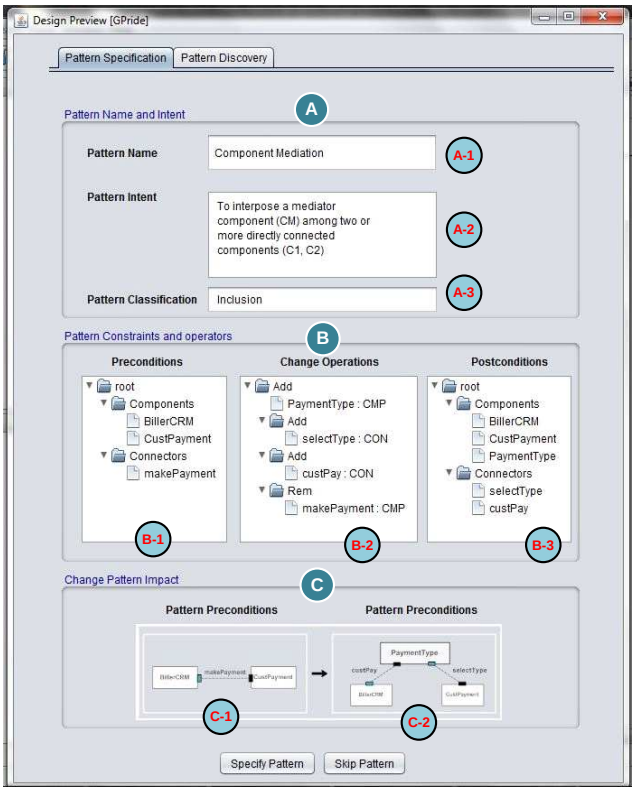


Figure 7: Screen-shot of Prototype for Change Pattern Specification.

A ***Pattern Name and Intent*** provide an overview of an individual pattern and its usage. The name and intent specify the primary role of a pattern during architecture evolution. The pattern author specifies the name and intent based on the pattern overview. For example, in Figure 7 the change pattern impact allows a user to select the pattern *name* Component Mediation and its *intent "to interpose a mediator component* C_M *among two or more directly connected components* (C_1, C_2)*"*. In addition, the user can also provide a *classification* type for the patterns. The pattern classification (Inclusion, Exclusion, Replacement) enables a logical grouping of related patterns based on the types of architectural changes that a group of patterns support. For example, the Component Mediation pattern can be classified as an Inclusion type pattern because it enables the inclusion of a new component C_M in an existing architecture model.

B ***Pattern Constraints and Operators*** provide an overview of the *constraints* on the architecture model that must be preserved before and after the pattern application as well as the

change operators that enable pattern-based change implementation. For example, in Figure 7 the constraints of Component Mediation pattern specify that before pattern application there must exist two components (C_1, C_2) interconnected using a connector X_1 as a precondition. In addition, the change operators enable the addition of the new component C_M and its connectors X_2, X_3 and removing its old connector X_1. Finally, the constraints specify that after the pattern application, a mediator component C_M has been successfully integrated in the architecture (post-condition).

C ***Change Pattern Impact*** provides an overview of the impact of a given pattern on the architecture model. It allows a user to see the changes a pattern enables before applying the pattern. The pattern provides a process-based change implementation by explicitly representing the conditions before, during and after the change implementation.

6.2 Overview of the Discovered Change Patterns

After pattern specification, we provide an overview of the discovered pattern in Figure 8. In Figure 8, we only provide a listing of all the patterns in terms of a) *pattern name and parameters*, b) *pattern intent*, c) *change operationalisation* and d) *pattern-based change impact*.

1. **Pattern Name and Parameters** A pattern name provides an identification of a pattern to its user. In addition, the parameters represent the affected architecture elements as a consequence of pattern application.
2. **Pattern Intent**[5] It represents a high-level pattern description in terms of the objective of pattern usage. For example, in Figure 8 the *Pattern Name* Component Mediation specifies the intent as a pattern that enables the integration of a mediator component C_M with directly connected component C_1, C_2.
3. **Change Operationalisation** It provides an operational syntax and semantics of architectural changes as a constrained composition of operators to enable architecture evolution.
4. **Pattern-based Change Impact** It represents the impact of change pattern on architecture models represented as the pre-conditions and post-conditions of change pattern.

7 Evaluation of Pattern-based Architecture Evolution

In this section, we demonstrate the applicability of change patterns to evolve a peer-to-peer system to a client-server architecture (in Section 7.1). We also evaluate the effects of the change patterns on efficiency of the architecture evolution process (in Section 7.2).

7.1 Pattern-based Architecture Evolution

A high-level architectural view of the peer-to-peer appointment system (P2P-AS) [42] is presented in Figure 9. Architectural components and connectors are represented inside configurations for modelling of P2P-AS system. The patterns presented in Section 6 (discovered from the EBPP [34] and 3-in-1 Telephone System [35] case studies) are applied and cross-validated to evolve a peer-to-peer architecture to a client-server architecture. Additional details about the component-connector view of P2P-AS architecture are provided in [42].

[5] The term *'pattern intent'* was first used in the GoF book to describe the primary objective of a pattern. However, nowadays, it is also common among pattern authors/users to use terms like pattern overview - pattern thumbnails or problem/solution-pairs.

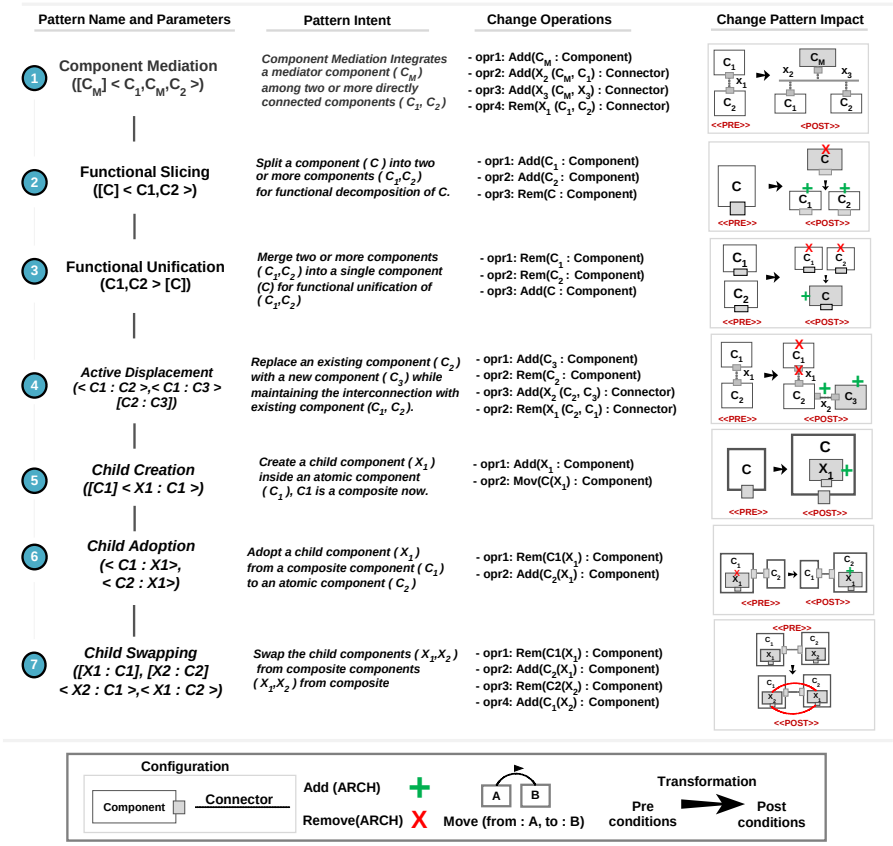


Figure 8: List of Discovered Architecture Change Patterns.

7.1.1 Source Architecture Model

The source architecture model of the P2P-AS system is presented in Figure 9 that consists of two configurations Client and Appointment Data. The Client configuration consists of an atomic component Appointment Client to request an appointment from the composite component Appointment Schedule in the Appointment Data configuration. The Appointment Schedule component is composed of Client Authentication component. The connector get Appointment enables the component interconnection.

7.1.2 Evolution Scenarios, Change Primitives and Patterns

We have presented the evolution scenario in Figure 9 (extracted from the P2P-AS case study [42]). We now provide a mapping of the evolution scenario (evolution problem) and the necessary change primitives and change patterns (as available solutions) in Table 3.

1. **Change Primitives** represent a collection of composite change operations to enable addition, removal and modification of individual components and connectors (Section 3, cf. Change

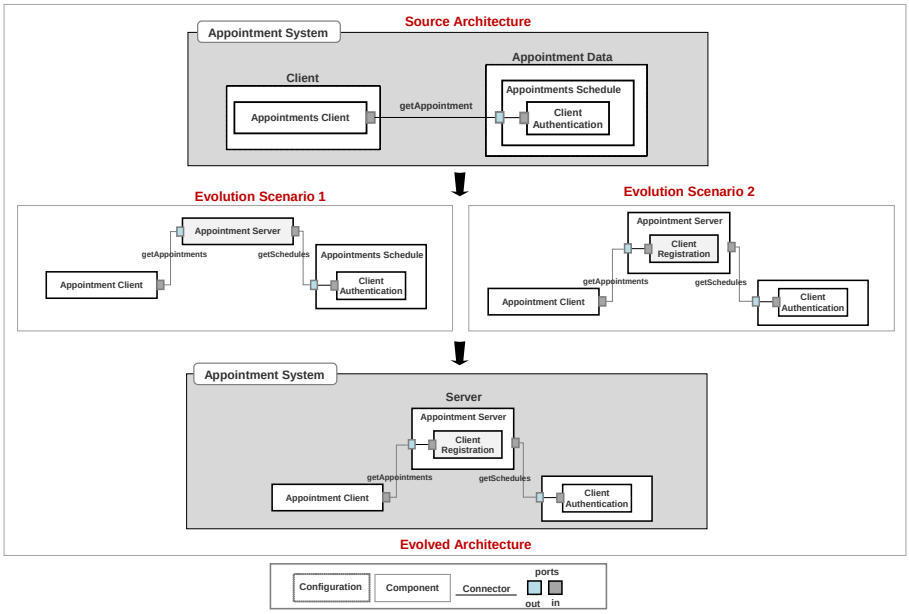


Figure 9: Source and Evolved Architecture with Architecture Evolution Scenarios.

Operators). For example, in Evolution Scenario 1 (Table 3) change primitive requires at-least a total of 4 change operations to integrate a mediator component in existing architecture. We only consider changes on architectural components and connectors omitting changes on ports and endpoints - it has already been explained that components must contain ports and connectors must contain endpoints (Section 3, Architecture Model).

2. **Change Patterns** abstract the change primitives and provides a generic and reusable operationalisation to enable architecture evolution. In contrast to primitives, change patterns provide a process-based implementation of architecture evolution. A pattern captures a reusable solution, its impact on architecture models (pattern pre/post conditions) and the necessary operations to enable evolution (detailed in Section 6). For example in Evolution Scenario 1 (Table 3), the Component Mediation pattern provides a reusable solution to primitives (ad hoc once-off change operations). We consider the specification of pre-conditions, a pattern and its post-conditions equivalent to a specification of 3 change operations.

7.1.3 Evolved Architecture Model

After the application of the change patterns to address the evolution scenarios, the evolved architecture model is presented in Figure 9. The evolved architecture model consists of a new configuration Appointment Server that is interposed between Client and Appointment Data. In the evolved architecture model, the composite component Appointment Server is used to handle the client request for appointments.

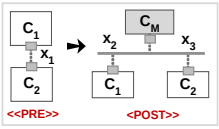
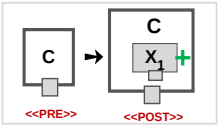
Evolution Scenario 1
To interpose the AppointmentServer component between the AppointmentClients and AppointmentSchedule components. The newly integrated Appointment Server component mediates between the client requests and appointment scheduling.
Change Primitives
CS-AS architecture is modified with addition of a new component AppointmentServer and two connectors (getAppointment, getSchedule) to enable mediation between Clients and Appointment components. $opr1 := ADD(AppointmentServer \in CMP)$ $opr2 := ADD(getAppointment((AppointmentClient, AppointmentServer) \in CMP) \in CON)$ $opr3 := ADD(getSchedule((AppointmentServer, AppointmentSchedule) \in CMP) \in CON)$ $opr4 := REM(getAppointment((AppointmentClient, AppointmentServer) \in CMP) \in CON)$
Change Pattern
$ComponentMediation([C_M] < C_1, C_M, C_2 >)$    To interpose a mediator component (CM) among two or more directly connected components (C_1, C_2).
Evolution Scenario 2
To create a child component ClientRegistration inside the AppointmentServer component. The newly added Client Registration component enables registration of individual clients on the server.
Change Primitives
CS-AS architecture is modified by creating the ClientRegistration component (atomic component) in Appointment Server (composite component) and a connector (register). $opr1 := ADD(ClientRegister \in CMP)$ $opr2 := ADD(register((ClientRegister, AppointmentClient) \in CMP) \in CON)$
Change Pattern
$ChildCreation([C] < X_1 : C >)$    To create a child component (X_1) inside an atomic component (C).

Table 3: A Summary of Evolution Scenarios, Change Primitives and Change Patterns.

7.2 Efficiency of Pattern-based Evolution

After presenting evolution scenarios and patterns to address these scenarios, we discuss the results of the evaluation. A summary of the evaluation results is presented in Table 4. In Table 4, we compare the efficiency of change implementation using change primitives and patterns with:

1. **Total Change Operations** To quantify the required efforts for change implementation, we count the number of change operators required for implementing a change and call this Total Change Operations (TCO). TCO is defined as *the total number of architecture change operations required to resolve an architecture evolution scenario.* For example, in Table 4 the TCO value for component integration is 4.

2. **Ratio of Change Operationalisation (Primitive vs Pattern)** Represents the ratio of change operators from pattern to primitive changes expressed as: $1 - (\frac{N_{TCO}}{E_{TCO}})$. N_{TCO} denotes the

number of change operations required by the patterns (N), E_{TCO} denotes the number of change operations required by the primitive (E). For an example, see Table 4,

Change Pattern		**Change Primitive**		**Efficiency Comparison**
Pattern Name	*TCO*	*Intent of Primitive*	*TCO*	N_{TCO}/E_{TCO}
Component Mediation	1	Integration of Components	4	25
Parallel Mediation	1	Integration of Components	3	33
Correlated Mediation	1	Integration of Components	8	12.5
Functional Slicing	1	Splitting of Components	3	33
Functional Unification	1	Merging of Components	3	33
Active Displacement	1	Replacement of Components	4	25
Child Creation	1	Composition of Components	4	25
Child Adoption	1	Move a Component	4	25
Child Swap	1	Swap a Component	4	25
	1		4.11	26%

Table 4: A Summary of Efforts for Change Primitives and Change Patterns.

Based on the summary of results in Table 4 we provide an overview of the comparative analysis for TCO for primitive and pattern-based changes. In contrast to patterns, primitive changes require between 3 and 8 change operations to implement a particular change. In addition, pattern-based changes provide a process-based overview of change implementation. The results suggest that:

Pattern-based changes take only 29% of change operations compared to primitive changes. However, pattern-based change does not support a fine-granular change representation.

The loss of granularity results in:

1. *Change Implementation at Higher Abstraction* - patterns with reusable but coarse-grained changes only provide generic changes that affect components and connectors. This abstraction do not support lower level changes changes at the component operations level, that are exposed at ports. In contrast, the change primitives supported with atomic and composite architectural changes support a fine granular change representation. The granularity of change implementation is a concern of source code changes [36] and not the architecture evolution.

2. *Structural Integrity of Architecture Model* - the granularity of architectural changes ensure that architectural integrity is preserved (components and their port, connectors have bindings). In our solution, architectural hierarchy is preserved with chnage operations that are abstracted in patterns.

8 Conclusions and Future Research

In this paper, we investigated architecture change logs - performing a post-mortem analysis of architectural evolution histories - to discover change patterns. In the context of architectural knowledge, the discovered patterns represent reusable knowledge and expertise that can be empirically discovered and reused to promote the notion of evolution-off-the-shelf in software architectures. The novelty of this research is a three-step process for evolution reuse that involves (a) *pattern discovery*, (b) *pattern specification*, and (c) *pattern instantiation*. To automate the pattern discovery process from change logs, we model the change log data as a graph and exploit graph

mining for change pattern discovery. Once the patterns are discovered, we provide a template-based specification to facilitate pattern documentation and its future reuse. Finally, we illustrate how pattern-based architecture evolution increases the efficiency of the architecture evolution processes and promote reusable change implementation.

Dimensions for Future Research include the evaluation of pattern discovery algorithms and pattern based reusability with more case studies. We need log data from different case studies as the developed algorithms can facilitate automated log mining to discover new patterns. The newly discovered patterns are assumed to support the composition and application of a *change pattern language* to support architecture evolution reuse. A pattern language provides a pattern collection such that the patterns in the language are formalised and interconnected. This means change patterns from the language could be selected and applied in a sequential fashion to support an incremental evolution. By incremental evolution we mean decomposing architectural evolution into a manageable set of evolution scenarios that could be addressed in a step-wise manner. Another interesting dimension is the possible identification and resolution of *change anti-patterns*. Change anti-patterns represent counter-productive and negative impacts of patterns on architecture models that can be mitigated to enhance the quality of architecture evolution.

References

1. T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer: Berlin Heidelberg, 2008.
2. K. H. Bennett and V. T. Rajlich, "Software Maintenance and Evolution: A Roadmap," in *Conference on the Future of Software Engineering*. ACM, 2000, pp. 73–87.
3. M. M. Lehman and J. F. Ramil, "Software Evolution: Background, Theory, Practice," *Information Processing Letters*, vol. 88, no. 1, pp. 33–44, 2003.
4. S. Lehnert, Q. Farooq, and M. Riebisch, "A Taxonomy of Change Types and its Application in Software Evolution," in *19th International Conference and Workshops on Engineering of Computer Based Systems*. IEEE, 2012, pp. 98–107.
5. A. Ahmad, P. Jamshidi, and C. Pahl, "Graph-based Pattern Identification from Architecture Change Logs," in *In Tenth International Workshop on System/Software Architecture*. Springer, 2012, pp. 200–213.
6. H. P. Breivold, I. Crnkovic, and M. Larsson, "A Systematic Review of Software Architecture Evolution Research," *Information and Software Technology*, vol. 54, no. 1, pp. 16–40, 2012.
7. D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, "Evolution Styles: Foundations and Tool Support for Software Architecture Evolution," in *Joint Working IEEE/IFIP Conference on Software Architecture, 2009 and European Conference on Software Architecture. WICSA/ECSA*. IEEE, 2009, pp. 131–140.
8. P. Jamshidi, M. Ghafari, A. Aakash, and C. Pahl, "A Framework for Classifying and Comparing Architecture-centric Software Evolution Research," in *17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. IEEE, 2013, pp. 305–314.
9. A. Ahmad, P. Jamshidi, and C. Pahl, "Classification and Comparison of Architecture Evolution-Reuse Knowledge - A Systematic Review," in *Journal of Software: Evolution and Process. DOI: 10.1002/smr.1643*. Wiley, 2014.
10. J. Cámara, P. Correia, R. De Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, "Evolving an Adaptive Industrial Software System to Use Architecture-based Self-adaptation," in *8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2013, pp. 13–22.

11. O. Le Goaer, D. Tamzalit, M. Oussalah, and A.-D. Seriai, “Evolution Shelf: Reusing Evolution Expertise Within Component-based Software Architectures,” in *32nd Annual IEEE International Computer Software and Applications, 2008. COMPSAC’08*. IEEE, 2008, pp. 311–318.

12. K. Yskout, R. Scandariato, and W. Joosen, “Change patterns: Co-evolving Requirements and Architecture,” *Journal of Software and Systems Modeling. DOI 10.1007/s10270-012-0276-6*, pp. 1–24, 2012.

13. H. Kagdi, M. L. Collard, and J. I. Maletic, “A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 19, no. 2, pp. 77–131, 2007.

14. T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining Version Histories to Guide Software Changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

15. R. Agrawal and R. Srikant, “Mining Sequential Patterns,” in *In Eleventh International Conference on Data Engineering, (ICDE’95)*. IEEE, 1995, pp. 3–14.

16. J. M. Barnes and D. Garlan, “Challenges in Developing a Software Architecture Evolution Tool as a Plug-Ins,” in *Proceedings of the 3rd Workshop on Developing Tools as Plugin-Ins (TOPI13)*, 2013, pp. 13–18.

17. C. Jiang, F. Coenen, and M. Zito, “A Survey of Frequent Subgraph Mining Algorithms,” *The Knowledge Engineering Review*, vol. 1, no. 1, pp. 1–31, 2012.

18. N. B. Harrison, P. Avgeriou, and U. Zdlin, “Using Patterns to Capture Architectural Decisions,” *IEEE Software*, vol. 24, no. 4, pp. 38–45, 2007.

19. X. Dong and M. W. Godfrey, “Identifying Architectural Change Patterns in Object-oriented Systems,” in *The 16th IEEE International Conference on Program Comprehension*. IEEE, 2008, pp. 33–42.

20. S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol, “Extracting Change-patterns from CVS Repositories,” in *Proceedings of the 13th Working Conference on Reverse Engineering*. IEEE Computer Society, 2006, pp. 221–230.

21. B. G. H. Tong, C. Faloutsos and T. Eliassi-Rad, “Fast Best-Effort Pattern Matching in Large Attributed Graphs,” in *13th ACM International Conference on Knowledge Discovery and Data Mining*, 2007.

22. M. Javed, Y. M. Abgaz, and C. Pahl, “Graph-based Discovery of Ontology Change Patterns,” in *Joint Workshop on Knowledge Evolution and Ontology Dynamics*. CEUR Workshop Proceeding, 2011, pp. 309–318.

23. L. P. V. H. Babar MA, Dingsyr T, *Software Architecture Knowledge Management: Theory and Practice*. Springer Heidelberg, 2009.

24. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-oriented Design*. Springer-Verlag LNCS, 1993.

25. N. R. Mehta and N. Medvidovic, “Composing Architectural Styles from Architectural Primitives,” in *9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, vol. 28, no. 5. ACM, 2003, pp. 347–350.

26. A. Ahmad, P. Jamshidi, and C. Pahl, "Graph-based Discovery of Architecture Change Patterns from Logs," *Technical Report: School of Computing, Dublin City University*, 2013. [Online]. Available: www.computing.dcu.ie\/~pjamshidi/PatternDiscovery.pdf

27. L. Yu, "Mining Change Logs and Release Notes to Understand Software Maintenance and Evolution," *CLEI Electron Journal*, vol. 12, no. 2, pp. 1–10, 2009.

28. P. Mohagheghi and R. Conradi, "An Empirical Study of Software Change: Origin, Acceptance rate, and Functionality vs. Quality Attributes," in *nternational Symposium on Empirical Software Engineering, ISESE'04*. IEEE, 2004, pp. 7–16.

29. V. Clerc, P. Lago, and H. van Vliet, "The Architect Mindset," in *Software Architectures, Components, and Applications*. Springer, 2007, pp. 231–249.

30. A. Ahmad, P. Jamshidi, M. Arshad, and C. Pahl, "Graph-based Implicit Knowledge Discovery from Architecture Change Logs," in *In Seventh Workshop on SHaring and Reusing Architecture Knowledge*. ACM, 2012, pp. 116–123.

31. N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

32. A. Ahmad and C. Pahl, "Pat-Evol: Pattern-drive Reuse in Architecture-based Evolution for Service Software," vol. 88. ERCIM News, 2012, pp. 200–213.

33. N. Lassing, D. Rijsenbrij, and H. van Vliet, "How Well can We Predict Changes at Architecture Design Time?" *Journal of Systems and Software*, vol. 65, no. 2, pp. 141–153, 2003.

34. EBPPCaseStudy, "Nacha - the electronic bill presentment and payment." [Online]. Available: www.nacha.org

35. 3-in-1 Phone System, "K3: Cordless Telephony Profile," *Bluetooth Specification Version*, vol. 1, 1999.

36. B. J. Williams and J. C. Carver, "Characterizing Software Architecture Changes: A Systematic Review," *Information and Software Technology*, vol. 52, no. 1, pp. 31–51, 2010.

37. P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-Level Modifiability Analysis (ALMA)," *Journal of Systems and Software*, vol. 69, no. 1, pp. 129–147, 2004.

38. H. Ehrig, U. Prange, and G. Taentzer, *Fundamental theory for typed attributed graph transformation*. Lecture Notes in Computer Science, Springer-Verlag, 2004.

39. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML Progress Report - Structural Layer Proposal," in *Graph Drawing*. Springer, 2002, pp. 501–512.

40. D. W. Brandes Ulrick, M. Gaertler, "Experiments on Graph Clustering Algorithms," in *11th Annual European Symposium on Algorithms*. Lecture Notes in Computer Science, 2007.

41. P. Clements, D. Garlan, R. Little, R. Nord, and J. Stafford, "Documenting Software Architectures: Views and Beyond," in *25th International Conference on Software Engineering, 2003*. IEEE, 2003, pp. 740–741.

42. N. S. Rosa, P. R. F. Cunha, and G. R. R. Justo, "An Approach for Reasoning and Refining Non-functional Requirements," *Journal of the Brazilian Computer Society*, vol. 10, no. 1, pp. 59–81, 2004.