Graph-based Discovery of Architecture Change Patterns from Logs

Aakash Ahmad, Pooayn Jamshidi, Claus Pahl

[ahmad.aakash—pooyan.jamshidi—claus.pahl]@computing.dcu.ie

Dublin City University

Faculty of Engineering and Computing, School of Computing

Technical Report

December 2012

Abstract

Modern software continuously evolves as a consequence of frequently varying requirements and changes in operational environments. Architecture-centric software evolution (**ACSE**) enables change in a system's structure and behavior while maintaining a global view of software to address evolution tradeoffs. Lehman's law of *continuing change* demands for long-living and continuously evolving architectures to prolong productive life and economic value of existing software. To tackle problems of recurring change, solutions for architectural maintenance and evolution must rely on reuse knowledge and expertise that promote evolution-off-shelf. State-of-the-art in ACSE suggests, although change patterns and evolution styles facilitate application of reuse knowledge to guide architecture evolution. However, there is a clear lack of research for a continuous acquisition of reuse knowledge in terms of *patterns*, *frameworks* and *processes* that can be discovered, shared and reused to guide ACSE. In this paper, we propose to exploit architecture change logs (**ACLs**) to perform post-mortem analysis of evolution histories and to discover reusable operationalisation and usage-determined architecture change patterns. The primary contribution lies with proposed algorithms - applied on change logs - to automate and customise pattern discovery process. An algorithm-based experimental investigation of change logs resulted in discovery of 7 architecture change patterns facilitating reuse in ACSE. We evaluate algorithmic efficiency in terms of discovering *exact* as well as *inexact* pattern instances when only central pattern feature suffice for its discovery. We provide a template-based specification to document patterns and demonstrate pattern-driven evolution of a peer-to-peer system towards a client server architecture. We highlight pattern discovery as a continuous process by acquiring data from different logs for mining new patterns over-time.

Keywords

Software Architecture Evolution, Evolution Reuse, Architecture Change Patterns, Pattern Discovery.

1 Introduction

Modern software continuously evolves as consequence of frequent changes in business and technical requirements and operating environments [1, 2]. Lehman's law of *'continuing change'* [2] states that " ...systems must be continually adapted or they become progressively less satisfactory". The primary challenges in supporting a continuous change [2] lies with i) acquisition and application of reusable solutions to address recurring evolution problems and ii) selection of an appropriate abstraction for software change implementation [3, 4]. To adress these challanges, we focus on acquistion of reusable solutions by discovering change patterns that promote reuse in architecture-centric software evolution (ACSE).

Architectural models proved successful in representing modules-of-code and their interconnections as high-level components and connectors to facilitate planning, modeling and executing software design and evolution at higher abstractions [5–7]. Our systematic reviews for state-of-the-research on ACSE suggests; solutions that tackle recurring evolution problems must rely on a continuous acquisition of evolution-centric knowledge that can be shared and reused to guide architecture change management. In particular, evolution-centric reuse-knowledge is defined as [3] : "[...] 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" [3,5].

Although, evolution styles [5, 6] and change patterns [8] promote *application of reuse knowledge* in architecture evolution, there is a clear lack of research on *acquisition of reuse knowledge* that involves a continuous discovery of new styles and patterns. In contrast to solution for pattern invention [5, 8], there is a need to develop and exploit experimental foundations to discover evolution patterns. In this research (the *PatEvol* project [9]) we aim to apply repository mining concepts [10] on software evolution histories to facilitate architecture change mining. We propose to exploit architecture change logs to investigate architecture change representation and discover architecture change patterns. We hypothesise that:

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

In a collaborative environment - involving multiple stakeholders - for architectural development and evolution, an architecture change log (**ACL**) provides a central repository to capture and maintain traces of architectural evolution [11, 12]. Evolution-centric knowledge in ACL includes but not limited to types and classification of change operations and dependencies among them along with architecture change patterns that can be discovered, shared and reused to enable 'evolution-off-the-shelf' in architectures. In this paper, we present patterns discovery from logs as a four-step process in Figure 1.

Step 1 *Capturing Architectural Change Instances in Logs* - In pattern discovery process, first we capture architectural change instances in logs. We exploit architecture-level modifiability analysis (ALMA) [13] for i) scenario-based elicitation of change instances from architecture evolution case studies [14, 15] and ii) record changes in logs [12, 16], detailed in Section 3 (input to step 2).

Step 2 *Graph-based Modeling of Change Log Data* - The change log data is formalised as an attributed graph [17] and represented using Graph Modeling Language (.GML) [18] format detailed in Section 3 (input to step 3). We apply sub-graph mining [19,20] - a formlised knowledge discovery technique - to identify recurring operationalisation and change sequences as change patterns.

Step 3 *Graph-based Discovery of Architecture Change Patterns* Once log-data is formalised as graphs, we develop algorithms for graph based pattern discovery [16, 19] that includes i) *Pattern Candidate Generation*, *Pattern Candidate Validation* and iii) *Pattern Candidate Matching*, as detailed in Section 4 (input to step 4).

Step 4 *Change Pattern Specification* Finally, we elaborate on discovered pattern instances and follow the guidelines in [21] to provide a template-based documentation for discovered patterns that enables pattern sharing and reuse, detailed in Section 5.

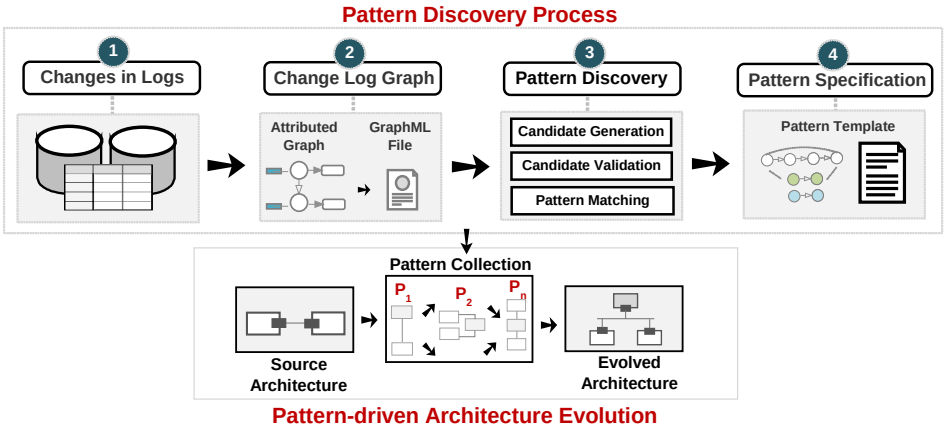


Figure 1. Process Overview for Log-based Change Pattern Discovery.

A case-study based demonstration highlights applicability of discovered patterns to guide architecture change management and evaluate pattern discovery algorithms. In the context of existing solution for pattern [8] and style-driven [5] evolution, the novelty lies with:

- Exploiting architecture change logs as a source of evolution-centric knowledge that enable post-mortem analysis of architecture evolution histories to discover reuse knowledge as architecture change patterns.
- We formalise log data as attributed graphs and exploit frequent sub-graph mining [19, 20] techniques to discover exact and inexact instances (patterns and their variants) that exist in a log.
- Pattern discovery algorithms executed on ACLs to address pattern discovery problem as a modular solution. Scalability of pattern-discovery process beyond manual analysis is supported with a prototype 'G-Pride' (Graph-based Pattern Identification) enabling automation and parameterised user intervention for pattern mining from logs.

2 Meta-model of Architecture Change Patterns

In change logs, we observed that operationalisation of individual changes represent a parameterised procedural abstraction [11]. This helps us to define change pattern as: *[. . .] "a generic, first class abstraction to support potentially reusable architectural change operationalisation and its execution"*. We formally express pattern-based architecture evolution as 5-tuple: $\mathsf{PatEvol} = < ARCH, OPR, CNS, PAT, COL >$ in Figure 2. Binary compositional relationships among change pattern (P) and its constituent element (E) are given as relation $\mathsf{P} \xleftarrow{relation} \mathsf{E} : < isContainedBy, isComposedOf, isAppliedTo, isConstrainedBy >$. For example, in Figure 2 the possible relation among a change pattern and change operators is expressed as: $\mathsf{Pattern} \xleftarrow{isComposedOf} \mathsf{Operators}$, a change pattern is composed of architecture change operations.

Definition 1. Architecture Model - Let ARCH represents the architecture model with $Configurations$ composed of $Components$ containing $Ports$ and $Connectors$ containing $Endpoints$ as:

$$\begin{aligned}
&\mathsf{ARCH} \sqsubseteq (Configuration \sqsubseteq Component \sqsubseteq Connector \sqsubseteq Port \sqsubseteq Endpoint),\\
&Configuration \equiv hasPart.(Component \sqcup Connector \sqcup Port \sqcup Endpoint)\\
&Component \equiv Arch \sqcap \exists hasComponent.Port\\
&Connector \equiv Arch \sqcap \exists hasConnector.Endpoint(Port_{source}, Port_{terget})
\end{aligned}$$

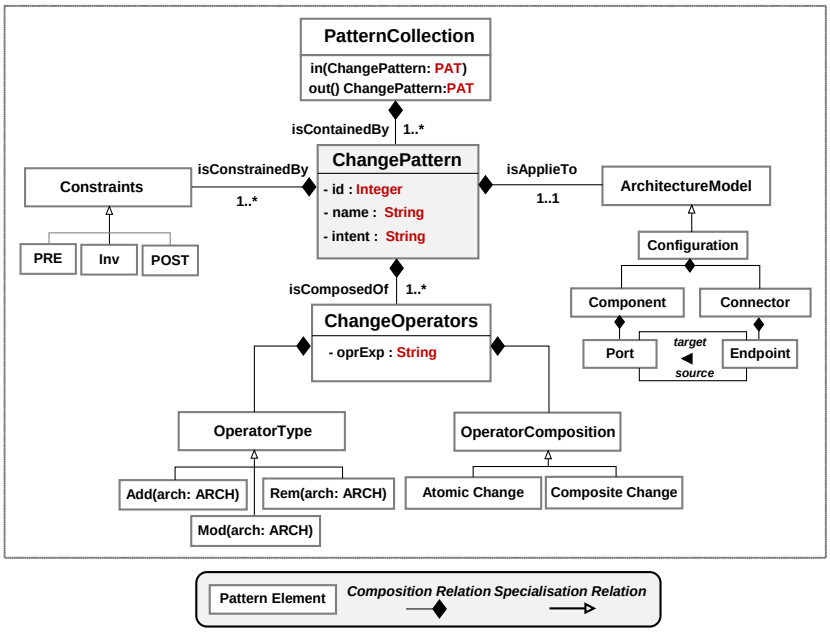


Figure 2. Meta-model for Composition of Architecture Change Pattern.

Architectural descriptions for component-based software architectures (**CBSA**) needs to be defined and constrained to achieve desired structure and semantics $\sqsubseteq$ (subsumption: architecture model), $\equiv$ (equivalence: instance of architecture elements) $\sqcup$ (disjunction: specialisation relation) $\sqcap$ (conjunction: composition relation). The relation *hasPart* describes composition of architectural configurations, while *hasComponent.Port* and *hasConnector.Endpoint* represent structural links from components to their ports and connectors to the endpoints (enabling interconnection among source and target ports).

Definition 2. Change Operator - Let an operation OPR represents adition, removal, modification type changes on architecture model ARCH (cf. Definition 1) expressed as:

$OPR(ARCH) :=< Add(arch \in ARCH), Rem(arch \in ARCH), Mod(arch \in ARCH) >$

- $Add(arch \in ARCH)$: addition of architecture (arch) in architecture model (ARCH).
- $Rem(arch \in ARCH)$: removal of element (arch) from architecture model (ARCH).
- $Mod(arch \in ARCH)$: modification of element (arch) in architecture model (ARCH).

Change operations represent a procedural abstraction to parameterise architectural changes that are fundamental to operationalising evolution. Our analysis of the change log [16] goes beyond basic types in [22] to specify a set of *atomic* and *composite* operations enabling structural evolution by adding (Add()), removing (Rem()) and modifying (Mod()) elements in architecture model (Definition 1).

Definition 3. Constraints - Let a constrained CNS representation of change operations OPR (Definition 2) on architecture model ARCH (Definition 1) expressed as:

$\text{PRE}[\text{OPR}(\text{arch} \in ARCH)] \xrightarrow{INV[OPR(arch \in ARCH)]} POST[OPR(arch' \in ARCH)]$

- $PRE[OPR(arch \in ARCH)]$: represents preconditions on change operations

- $INV[OPR(arch \in ARCH)]$: represents invariants on change operations
- $POST[OPR(arch' \in ARCH)]$: represents post-conditions on change operations

During change operationalisation pre-conditions represent the context of architectural model before change execution. It represents complete or partial source architecture model that is evolved towards a target model. In addition, any change in the architectural structure must maintain the correctness of architectural invariant. Any violation of the architectural invariant results in an invalid instance of an architecture element. Finally, post-conditions represents evolved architecture model as a consequence on change operationalisation on architecture elements.

Definition 4. Change Pattern - Let a Pattern (PAT) represents a recurring, constrained (CNS) composition of change operationalisation (OPR) on architecture model (ARCH) expressed as:

$$\text{PAT}_{<name,intent>} : PRE[OPR(arch \in ARCH)] \xrightarrow{INV[OPR(arch \in ARCH)]} POST[arch' \in ARCH]$$

It represents a first-class abstraction that can be operationalised and parameterised to support potentially reusable architectural change execution. A pattern enables a process-oriented approach to architecture change management describing the situation before and after the change (constraints), along with the steps needed to make the transition (operations). In addition a pattern's *name* and its *intent* introduces pattern vocabulary. Pattern vocabulary provides an abstract view of problem-solution map (change operations, their impact and trade-offs) captured by pattern name and intent.

Definition 5. Pattern Collection - Let $pat_i, pat_j, \ldots, pat_n \in PAT$ represent discovered pattern instances conforming to the meta-model PAT in Figure 2, we express a COL collection of architecture change patterns as: $COL_{(in,out)} =< pat_i, pat_j, \ldots, pat_n >$

- $in(pat_i \in PAT)$ - is a function for storage of a pattern instance $pat_i \in PAT$.
- $out() : pat_i \in PAT$ - is a function for retrieval of a pattern instance $pat_i \in PAT$.

A pattern collection is essentially a repository infrastructure to maintain discovered pattern instances. We follow the guidelines in [21] to develop an architecture change pattern template that provides a structured document to represent the name, intent to promote pattern as a solution that can be queried and retrieved. In addition, a high-level solution and consequences for each pattern instance (PAT) are documented in the template that includes operationalisation (OPR), constraints (CNS) and the affected architecture elements (ARCH).

3 Graph-based Modeling of Architecture Change Log Data

In this section, we model architecture change log data [12] as a log graph [17,20] (for graph-based pattern discovery [19]) and utilise ALMA method [13] for scenario-based analysis of architectural changes in logs. We follow a case study-based approach and analyse architectural evolution case for an i) Electronic Bill Presentment and Payment System (EBPP) [14] and ii) 3-in-1 Phone system [15] - EBPP case study used as running example. The data in change log can be continuously updated based on the availability of evolution-centric information for new case studies to promote pattern discovery as a continuous process.

3.1 Types of Data in Architecture Change Logs

Source to target architecture evolution includes a number of architectural change instances by means of change operations on architecture elements. If these changes are not captured, this results in the loss of architectural change data we refer to as the *evaporation of evolution-centric information*. In contrast,

capturing each individual architectural change enables a fine-granular representation of architecture evolution history in a repository infrastructure (ACLs) as *absorption of evolution-centric information* [3].

Definition 6. Architecture Change Log - Let OPR represent a change operation on architecture model arch ∈ ARCH, Architecture Change Log ACL is expressed as:

$$ACL = (OPR_1(arch \in ARCH) \prec OPR_2(arch \in ARCH) \prec \ldots \prec OPR_N(arch \in ARCH))$$

ACL denotes a sequential collection of change operations, $\prec$ is a sequencing operation between consecutive change operations (such that OPR_1 precedes OPR_2 and so on). However, during evolution it is possible that certain change operations can be applied in a *sequential* fashion, while other operations may be applied in *parallel* to each other. Technical details to analyse the extent to which architecture change operations are dependent (*sequential*) or independent (*commutative*) of each other are already detailed in our previous work [16]. We conclude that, only sequential changes that are frequent in log represent potential change patterns. Once sequential operations are recorded in ACL, change log data is classified as *Change Data (CD)* and *Auxiliary Data (AD)* as represented in Figure 3.

1. *Change Data (CD)*: contains the core information about individual change operations in the log expressed as $CD = (ChangeID, OPR, ARCH)$ representing change $id(opr_1 \prec opr_2 \prec \ldots \prec opr_n)$ along with change operations on architecture elements. For example in Figure 3 b, change data represents change id as opr_1 to add a new component PaymentType inside Payment configuration.
2. *Auxiliary Data (AD)*: is represented as $AD = (UserID, TimeStamp, ChangeIntent, SytemID)$ 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 3. 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 log.

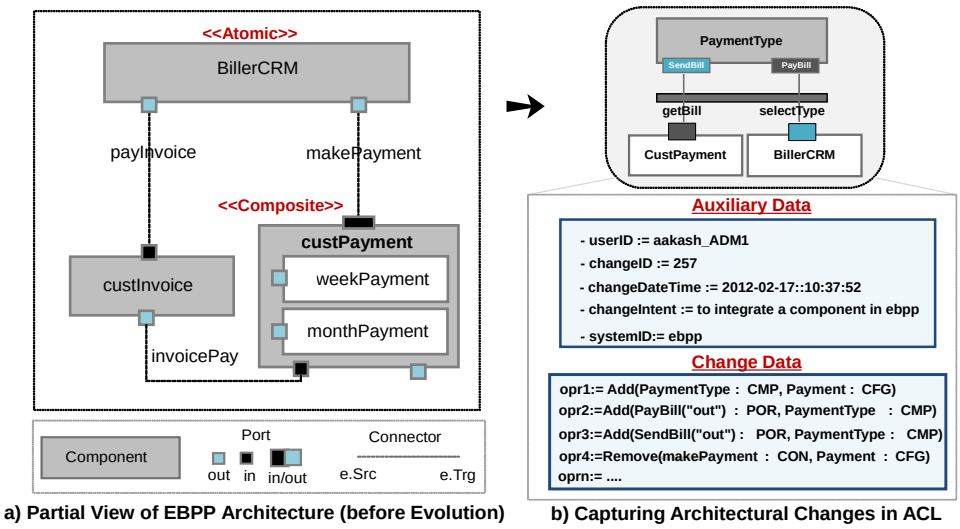


Figure 3. a) Partial Architectural View for EBPP and b) Capturing Change Instances in ACL

We ALMA [13] for evolution scenario elicitation and analysis of EBPP evolution as follows.

A *Scenario Selection - Component Integration*: In the existing functional scope of the case study (Figure 3a) the company charges its customer with full payment of customer bills in-advance to deliver 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 3b, 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 connected components BillerCRM and CustPayment.*

B *Scenario Evaluation - Analysing Architectural Changes*: In selected scenario above, existing EBPP architecture is modified with addition of new components PaymentType (and corresponding ports) and two connector getBill and selectType to mediate customer billing and payments in Figure 3b. This results in recording individual change operations in the log (**change data**) along with the intent, time and effects of change (**auxiliary data**) as illustrated in Figure 3b.

C *Results Interpretation - Impacts of Architectural Changes*: We nalyse source architecture (as pre-conditions of evolution) Figure 3a, the change instances (as change operations) applied on source architecture to achieve the evolved architecture (as post-conditions of evolution) Figure 3b.

3.2 Creating Change Log Graph

In this section, we focus on formalising change instances in the log as an *attributed graph (AG)* with nodes and edges typed over an *attributed typed graph (ATG)* [17]. Please note, an AG in Figure 4 represents a meta-graph to model change log data as an ATG that represents an instance-graph AG in Figure 5. It is vital to mention that although creating a graph form change log data requires additional efforts (mapping each change operation from logs to graph node and edge). However once a change log graph is created our evaluation (in Section 5) shows graph-based searching and retrieval of log data is faster and efficient that querying the traditional file-based log systems. An inherent benefit in a graph-based modeling of log data lies with exploitation of sub-graph mining - a formalised graph mining technique - whereas recurring sub-graphs in log graph can be discovered as frequent change patterns.

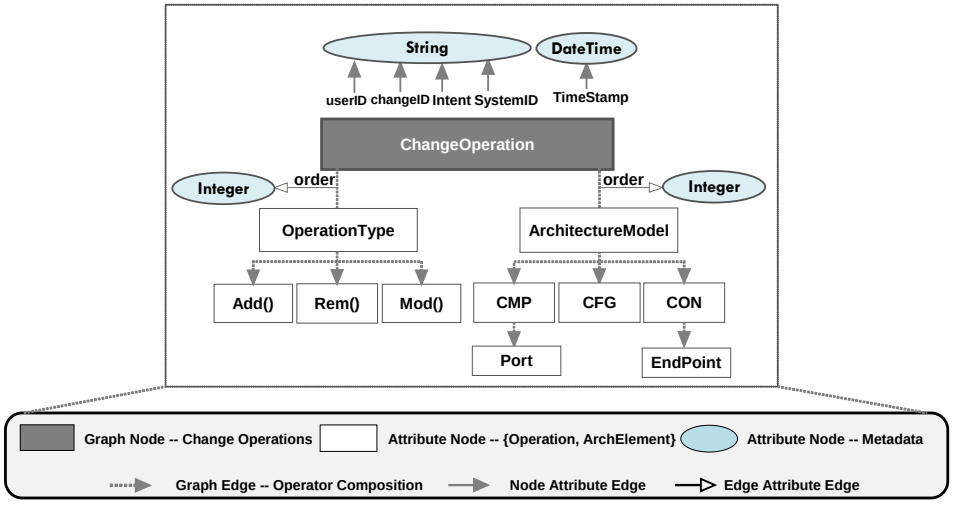


Figure 4. Attributed Typed Graph Model to Formalised Architecture Change Log Data

Definition 7. Architecture Change Log Graph - A collection of change operations (Definition 2) from log ACL (Definition 4) are expressed as an attributed change log graph G_{ACL}:

$G_{ACL} = \langle N_G, N_A, E_G, E_{N_A}, E_{E_A} \rangle$

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

The attributed graph morphism M from an instance graph AG (Figure 4) to its meta-graph ATG (Figure 5) is expressed as $M : AG \rightarrow ATG$. A collection of change operations Definition (2) from log ACL Definition (5) are expressed as an attributed change log graph G_{ACL}illustrated in Figure 4:

1. $N_G = \langle n_g^i | i = 1, \ldots, m \rangle$ 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) [11, 16].
2. $N_A = \langle n_a^i | i = 1, \ldots, m \rangle$ represents a set of attribute nodes for graph nodes (N_G). Attribute nodes are of two types, i) attribute nodes that represent auxiliary data (e.g. userID, changeID, TimeStamp etc.) and ii) 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 change log graph.
3. $E_G = \langle n_g^i | i = 1, \ldots, m-1 \rangle$ represents a set of graph edges that connects two graph nodes N_G. The graph edges ($e_g \in E_G$) represents the applied sequence of change operations (OPR) applied on the architecture model (ARCH). The term $i = 1, \ldots, m$ represents total graph edges in log graph.
4. $E_{N_A} = \langle e_{na}^i | i = 1, \ldots, p \rangle$ 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 a architecture change graph.
5. $E_{E_A} = \langle e_{ea}^i | i = 1, \ldots, q \rangle$ 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.

3.3 Creating Architecture Change Session Graph

The change session graph enables extraction of a subset of all the change instances in the log based on intent, scope or time of architectural changes. Continuing with earlier example (addition of a PaymentType component, cf. Figure 3), in Figure 5 we present a partial view of the change session graph that is an instance of change graph in Figure 4.

Architecture change session is calculated based on an interval (endTime - start Time) as all the changes between time stamp endTime(17-02-2012::10:41:35) and startTime(17-02-2012::10:37:52). Session-based change mining is helpful in analysing a subset of all the changes from logs (time-interval of architectural change defines change subset in this example).

In Figure 5, attributed graph morphism $t : AG \rightarrow ATG$ is defined over graph nodes with $t(ATG) = AG$ that results in 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 eg for source and target nodes (257, 258, 263, 264) representing the applied sequence of change operations. The change log instances as an attributed graph are mapped to the Graph Modeling Language (.GML)[1] [18] format.

[1] A Sample of Change Log Graph presented using GRAPHML format is available at: `www.computing.dcu\~aaakash\loggraph.html`

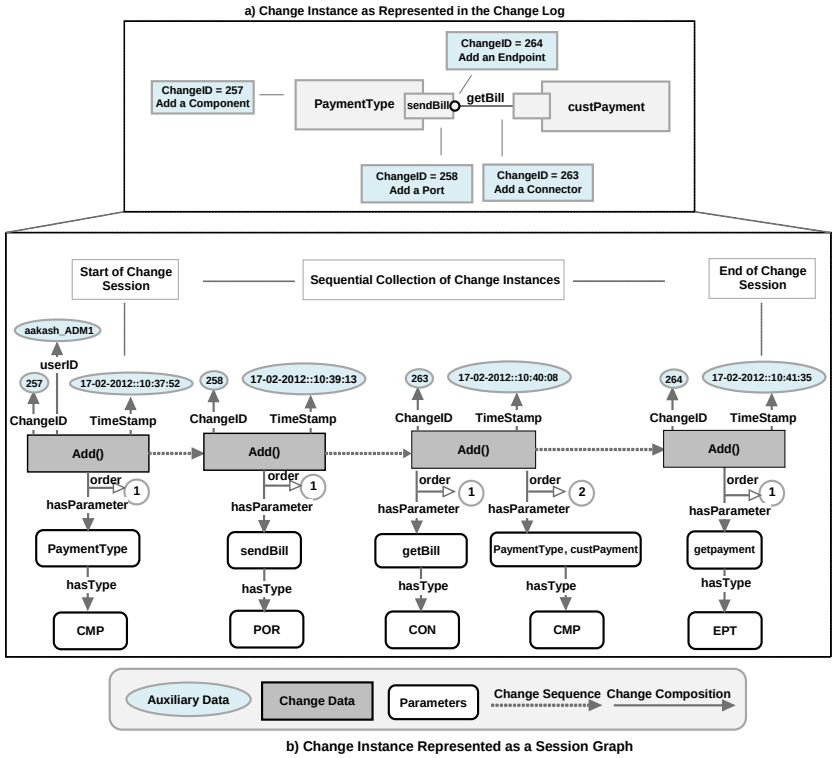


Figure 5. Change Instances as an Attributed Graph (typed over ATG in Figure 4).

4 Change Operation Sequences and Pattern Discovery

In this section we elaborate on application of sub-graph mining to discover recurring patterns as frequent sub-graphs in change log graph. First, in Section 4.1 we present types and properties of architectural change sequences that help us to determine recurring sequences in logs as potential change patterns. In Section 4.2, we provide algorithmic details for graph-based change pattern discovery.

4.1 Properties and Types of Architectural Change Operation Sequences

In Section 2, we elaborated on atomic and composite architectural changes to operationalise evolution in terms of addition, removal or modification of architecture element. In contrast, an architecture change sequence represents *a sequential collection of atomic and composite changes to perform higher-level change operations in terms of integration, composition, replacement etc. of components and connectors in architectural configurations* in Table 1. We present two architecture change sequences (S_1 and S_2) extracted from the change log, S_1 is used from the running example also presented in Figure 5 (change log graph).

1. **Change Sequence 1 (S_1) - Component Integration**: This sequence represents integration of a PaymentType component among directly connected components CustPayment and BillerCRM already explained (cf. Figure 5).

2. **Change Sequence 2 (S_2) - Component Replacement**: This sequence represents replacement of old component WeekPayment with MonthPayment in Billing configuration. It enables removal of a

Table 1. Change Sequences (S_1 and S_2) as Extracted from the Change Log

Sequence (S_1)		
CID	Change Operation	Change Impact
257	Add(PaymentType ∈ CMP)	Payment ∈ CFG
258	Add(sendBill ∈ POR)	PaymentType ∈ CMP
263	Add(getBill ∈ CON)	(CustPayment, PaymentType) ∈ CMP
264	Add(getPayment ∈ EPT)	getBill ∈ CON
Sequence (S_2)		
CID	Change Operation	Change Impact
715	Rem(WeekPayment ∈ CMP)	Billing ∈ CFG
716	Add(MonthPayment ∈ CMP)	Billing ∈ CFG
717	Rem(payWeekly ∈ CON)	(CustBill, WeekPayment) ∈ CMP
718	Add(payMonthly ∈ CON)	(CustBill, WeekPayment) ∈ CMP

component payWeekly (CID:=715) from Billing Configuration and add a new component payMonthly (CID:=716)) in same configuration. Addition of new component follows removal of the old connector payWeekly (CID:=717) that precedes addition of the new connector payMonthly (CID:=718) between CustBill and MonthPayment components.

Change sequences allow us to abstract the individual changes into a collection of recurring change operations representing potential patterns determined by properties of change sequences detailed below. In order to exemplify the properties of change sequences, we used sequences S_1 and S_2 from Table 1.

Property I - Type Equivalence (TypeEqu) refers to the equivalence of two change operations given by the utility function $\mathsf{TypeEqu}(OPR_j(arch_i \in ARCH), OPR_k(arch_j \in ARCH)) : returns < boolean >$. 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, mapping S_1 and S_2 the change operation Add(PaymentType ∈ CMP) is only equivalent to Add(MonthPayment ∈ CMP) and Add(getBill ∈ CON) is only equivalent to Add(payMonthly ∈ CON) $TypeEqu(S_{1_{257}}, S_{2_{716}})$ and $TypeEqu(S_{1_{263}}, S_{2_{718}})$ returns true, false in any other case from Table 1.

Property II - 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 $\mathsf{LenEqu}(S_1, S_2)$: returns $< integer >$. Therefore, the length equivalence of two change sequences S_1 and S_2 is determined by the numerical value (0 imples $S_1 == S_2$, -n implies $S_1 < S_2$ by n nodes and +n implies $S_1 > S_1$ by n nodes). For example, in Table 1 the length of $S_1 == S_2$ so $TypeEqu(S_1, S_2)$ returns 0.

Property III - 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 at different times. For example the order of change operations in S_2 can be represented as any of the given sequence $S2 = S_{2_i}(715, 716, 717, 718)(OR)S_{2_j}(715, 717, 716, 718)(OR)S_{2_k}(716, 718, 715, 717)$. The semantics and impact of change operation remains the same even if sequencing of change operations is varied (a component is replaced by applying either S_{2_i} or S_{2_j} or S_{2_k}. It is given by the function $OrdEqu(S_1, S_2)$: returns $< boolean >$. We distinguish among four types of sequences in Table 2.

A summary of functions and their return types to identify different change sequences is presented in Table 2. Excat and inexact sequences are fundamental to discovering change patterns (exact and in-exact graph matching).

- *Exact Sequence*: Two given sequences are exact sub-sequences if they match on operational types, length equivalence and the ordering of the change operations. In Table 1, S_1 and S_2 are not the

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

Sequence Type	**TypeEquv**($S_{1_{OPR}}$, $S_{2_{OPR}}$)	**LenEquv**(S_1, S_2)	**OrdEquv**(S_1, S_2)
Exact Sequence	true	0	true
Inexact Sequence	true	0	false
Partial Exact Sequence	true	$\pm n$	true
Partial inexact Sequence	true	$\pm n$	false

exact sequences, although sequence length is same (four change operations) however, the order and type of all operations do not match.

- *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_1 and S_2 are not the inexact matching sequences, although sequences have similar length $S_1 == S_2$, however type for operations do not match. An example of inexact sequences is illustrated above where S_{2_j} and S_{2_k} is an inexact equivalent of S_{2_i} as sequence length and types of change operations in both S_{2_i} or S_{2_j} are equivalent, however the order of their change operations is distinct.

- *Partial Exact Sequence*: Two given sequences S_1 and S_2 are partially exact such that (if $n > 0$ than $S_2 \subset S_1$, or if $n < 0$ than $S_1 \subset S_2$) - however, the types and ordering of the change operations in the matched sequences must be equivalent. In Table 1 S_1 and S_2 are not partial exact matching sequences as the order of operations in both the sequences does not match.

- *Partial Inexact Sequence*: Two given sequences S_1 and S_2 are partial and inexact if (if $n > 0$ $S_2 \subset S_1$, or if $n < 0$ than $S_1 \subset S_2$); 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_1 and S_2 are not partial inexact match.

4.2 Change Pattern Discovery Algorithms

Once change log data is formalised as an attributed graph, the solution to pattern discovery problem lies with application of sub-graph mining approaches [19, 20] on chnage log graph. Graph-based pattern discovery lies with mining recurrent sequences (cf. Section 4.1) of change operations that is equivalent to discovering sub-graphs which occur frequently over change graph G_{ACL}. In this section, we introduce the pattern discovery problem as a modular solution and present pattern discovery algorithms following a 3-step process illustrated in Figure 6.

1. *Candidate Generation*: A pattern candidate (P_C) is a sequence of change operations in the graph (G_{ACL}) that represents a potential change pattern (PAT) depending on its occurrence frequency $Freq(P_C)$ in change graph, such that $P_C, PAT \subseteq G_{ACL}$ illustrated in Figure 6a - Algorithm I.

2. *Candidate Validation*: The candidate validation is a significant step to eliminate the false positives, i.e; candidates that represent potential patterns but their application as change pattern may result in violating the invariants of architecture model, illustrated in Figure 6b - Algorithm II.

3. *Candidate Validation*: Once the candidates are validated, we utilise graph matching to (that compares graph nodes as change operations) to discover recurring candidate sequences (P_C) as patterns (PAT), in Figure6c - Algorithm III.

In Table 3, we provide a list of variables that facilitate parameterisation of algorithms for pattern discovery process. In Table 4, we outline a number of utility functions that are frequently used to maintain the modularity of the pattern discovery process.

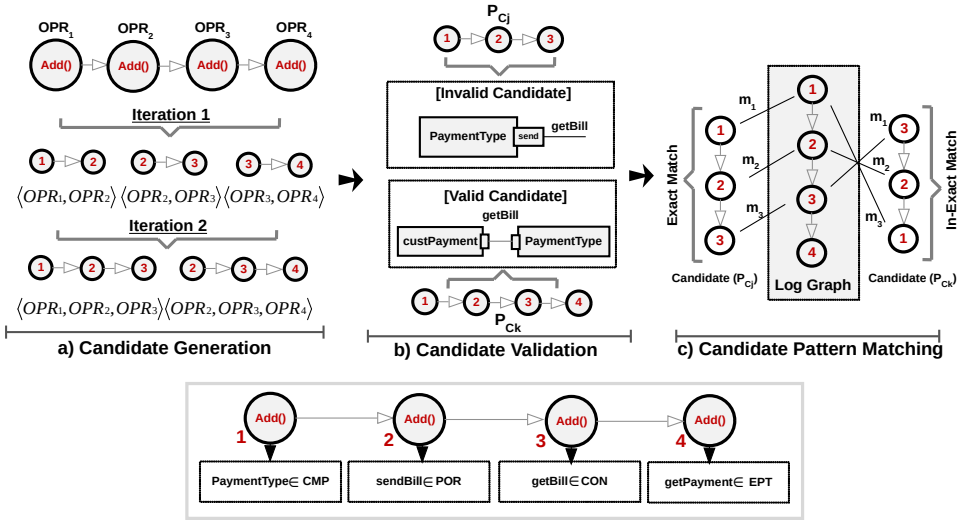


Figure 6. Overview of 3-Step Graph-based Pattern Discovery Process

Table 3. Parameters for Graph-based Pattern Discovery process

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

Table 4. A List of Utility Methods for Pattern Discovery

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	Bi-jective node matching based on $TypeEquv()$ (Section 4.1)
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}

4.2.1 Algorithm I - Candidate Generation

As the initial step of pattern discovery process, candidate generation aims at generating a set of pattern candidates P_C from architecture change graph G_{ACL}, as illustrated in Figure 6a.

i **Input**: is a user specified change graph G_{ACL} along with minimum $minLen(P_C)$ and maximum $maxLen(P_C)$ candidate lengths $minLen(P_C)$: 2 and $maxLen(P_C)$: 3 in Figure 6a).

ii **Process**:

We apply graph clustering approach [23] on G_{ACL} to create graph clusters representing sub-graphs as pattern candidates in Figure 6a. 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 3. The size $Len(P_C)$ of a cluster (P_C) represents the total nodes in a cluster that ultimately represents the number of change operations in P_C. For example, in Figure 6a the user specifies $minLen(P_C) : 2$ and $maxLen(P_C) : 3$, in first iteration candidates are generated such that length of each candidate is two nodes with next iteration each candidate having three nodes. The generation of pattern candidates $PC_1, \ldots, PC_N$ based on graph clustering [23] is expressed:

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

The process starts at graph root with selection of a single node and enumerating the temporary candidate list with adjacent node concatenation. Based on minimum and maximum candidate length: $buff(P_C) = \langle pc_1(OPR_1, OPR_2), cp_2(OPR_2, OPR_3), \ldots, pc_5(OPR_2, OPR_3, OPR_4) \rangle$ is created as a temporary candidate length presented (Line 1 - 13). To avoid this exhaustive candidate list, the candidates in $buff(P_C)$ are iteratively matched to find specific candidates that occur atleast more than once in G_{ACL}. We use the Breadth First Search (BFS) [19] over G_{ACL} with our matching function $nodeMatching(n_i; n_j)$ (Table 4) : $n_i.OPR \xrightarrow{match} n_j.OPR \wedge n_i.ARCH \xrightarrow{match} n_j.ARCH$ to generates final candidate list: $List(P_C)$ (Line 14 - 23). We ensure each candidate $pc_i \in List(P_C)$ is validated through candidateValidation($cp : G_{ACL}$) (Line 18, detailed in Section 4.2.2).

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

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 candLength ≤ maxLen(P_C) do
 4     maxCandidates ← G_ACL.size() - candLength
 5     while root ≤ maxCandidates do
 6         for node ← 0 to node ≤ candLength do
 7             buff(P_C)_node ← G_ACL(node + root)
 8             node ← node + 1
 9             root ← root + 1
10         end
11         candLength ← candLength + 1
12     end
13 end
14 List(P_C) ← φ {List of final candidates}
15 for tempCand ← 0 to tempCand ≤ buff(P_C).Length() do
16     for cand ← 1 to cand ≤ buff(P_C).Length() do
17         if buff(P_C)_tempCand.Length() == buff(P_C)_Cand.Length() then
18             if nodeMatching(tempCand, cand) == true and candidateValidation(cand) == true then
19                 List(P_C)_tempCand ← buff(P_C)_cand
20             end
21         end
22     end
23 end
24 return
```

4.2.2 Algorithm II - Candidate Validation

During candidate generation, there may exist some false positives in terms of candidates that violate the structural integrity (invariants) of architecture model when identified and applied as patterns. For example, in Figure 6b the candidate P_{C_j} represents three change operations $\{Add(PaymentType \in CMP)$

$\prec Add(sendBill \in POR) \prec Add(getBill \in CON)$ that adds a component, its port and a connector. However, the connector do not provides interconnection among source and target ports (an orphan connector). Therefore, it is vital to eliminate the candidate pattern P_{C_j} that may violate architectural integrity (invalid candidate). In contrast, the candidate P_{C_k} represents four change operations and provides interconnection among component ports in Figure 6b, referred as a valid candidate. We eliminate invalid candidates through validation for each generated candidate *pc* against architectural invariants (cf. Definition 1) before pattern matching with algorithmic details below:

i **Input**: is a candidate $c_{p_i} \in P_C$, $P_C \subseteq G_{ACL}$ (called from candidateGeneration() - Line 18).

ii **Process**: includes look-up into invariant table in terms of validating configuration of architecture elements in the candidates as $lookUp(arch \in ARCH) : \forall cfg \in CFG \exists con_i((cmp_i; cmp_j) \in CMP) \in CON$(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, while orphaned connectors have no associated component, indicated by Boolean type false.

iii **Output**: is a Boolean value indicating either valid (true) or invalid (false) candidate *cp*.

Algorithm 2: candidateValidation()

```
   input  : cp ∈ G_ACL
   output: boolean(true/false)
 1 isValid ← false
 2 iteration : for node ← 0 to node ≤ pc.Length do
 3     if lookUp(pc.node.ARCH) == true then
 4         isValid ← true
 5     else
 6         isValid ← false
 7         break iteration
 8     end
 9 end
10 return isValid
```

4.2.3 Algorithm III - Candidate Pattern Matching

Once the candidates are validated, the last step involves candidate pattern matching based on a user specified frequency threshold $Freq(P_C)$ for P_C in G_C. This means user specify that 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 isomorphism to match graph nodes of P_C and G_{ACL} iteratively.

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

ii **Process**: includes retrieving each candidate from $List(P_C)$ and finds its exact or possible inexact instance in G_{ACL}. In any given match from P_C to G_{ACL} the number of nodes must be equal (Line 11). We exploit the change sequence properties in Table 2 to specify: If and only if all the nodes in candidate match the corresponding nodes in change graph we refer to this as P_C is isomorphic to G_{ACL}, ($P_C \simeq G_{ACL}$) as relation:

$$nodeMatching(P_C, G_{ACL}) = \begin{Bmatrix} \langle C_{P_1}(OPR_1, OPR_2) & \cdots & C_{P_n}(OPR_i, OPR_j, \ldots, OPR_k) \\ \vdots & \ddots & \vdots \\ G_{ACL}(OPR_1, OPR_2, \ldots, OPR_N) & & G_{ACL}(OPR_1, OPR_2, \ldots, OPR_N) \end{Bmatrix}$$

- *Exact Match* : There must exist bijective mapping among types of change operator and the type of architecture element in attributed nodes with exactMatch($nodeMatching(n_i; n_j)$)$[\forall(i,j) = 1 \dots N]$ that utilises the $nodeMatching(n_i, n_j)$ 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 6c. In addition the ordering of matching nodes from $List(P_C)$ to G_{ACL} must be same. If such an exact instance is found, candidate's frequency is incremented and matching is repeated (Line 13, 14), otherwise:

- *Inexact Match* : The order of matching nodes from $List(P_C)$ to G_{ACL} is not always same. In this case, $inexactMatch(nodeMapping(n_i; n_j))[\forall(i \rightarrow j) = 1 \dots N]$ is called that utilises the $nodeMatching(n_i, n_j)$ method 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 6c. The candidate's frequency is incremented and matching is repeated until leaf node (Line 16, 17).

iii **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 user specified frequency threshold: $freq(PAT) \geq Freq(P_C)$.

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     while root ≤ G_ACL.getLeaf() do
 6         exactMatch ← 0
 7         inexactMatch ← 0
 8         for node ← root to node ≤ List(P_C)_cand .Length() do
 9             gCand(node ← node + 1) ← (root ← root + 1)
10         end
11         if List(P_C)_cand.Length() == gCand(root).Length() then
12             for node ← root to node ≤ List(P_C)_cand .Length() do
13                 if match(List(P_C)_cand.node, gCand(root).node == true) then
14                     exactMatch ← exactMatch + 1
15                 else
16                     if inexactmatch(List(P_C)_cand.node, gCand(root).node == true) then
17                         inexactMatch ← inexactMatch + 1
18                     end
19                 end
20             end
21         end
22         if exactMatch == List(P_C)_cand.Length() or inexactMatch == List(P_C)_cand.Length() then
23             freq++
24         end
25     end
26     if freq ≥ Freq(P_C) then
27         pList(PAT, Freq(PAT)) ← (List(P_C)_cand, freq)
28     end
29 end
```

5 Results and Discussions

A list of discoverd pattern instances is presented in Figure 7, please refer to [24] for addition details about individual pattern instances. We demonstrate patter-driven architecture evolution in Section 5. In this section, first we demonstrate role of change patterns in supporting pattern-driven reuse in architecture evolution (Section 5.1). Secondly, we perform an experimental evaluation to analyse accuracy and effects of user-based customisation of pattern discovery algorithms (Section 5.2).

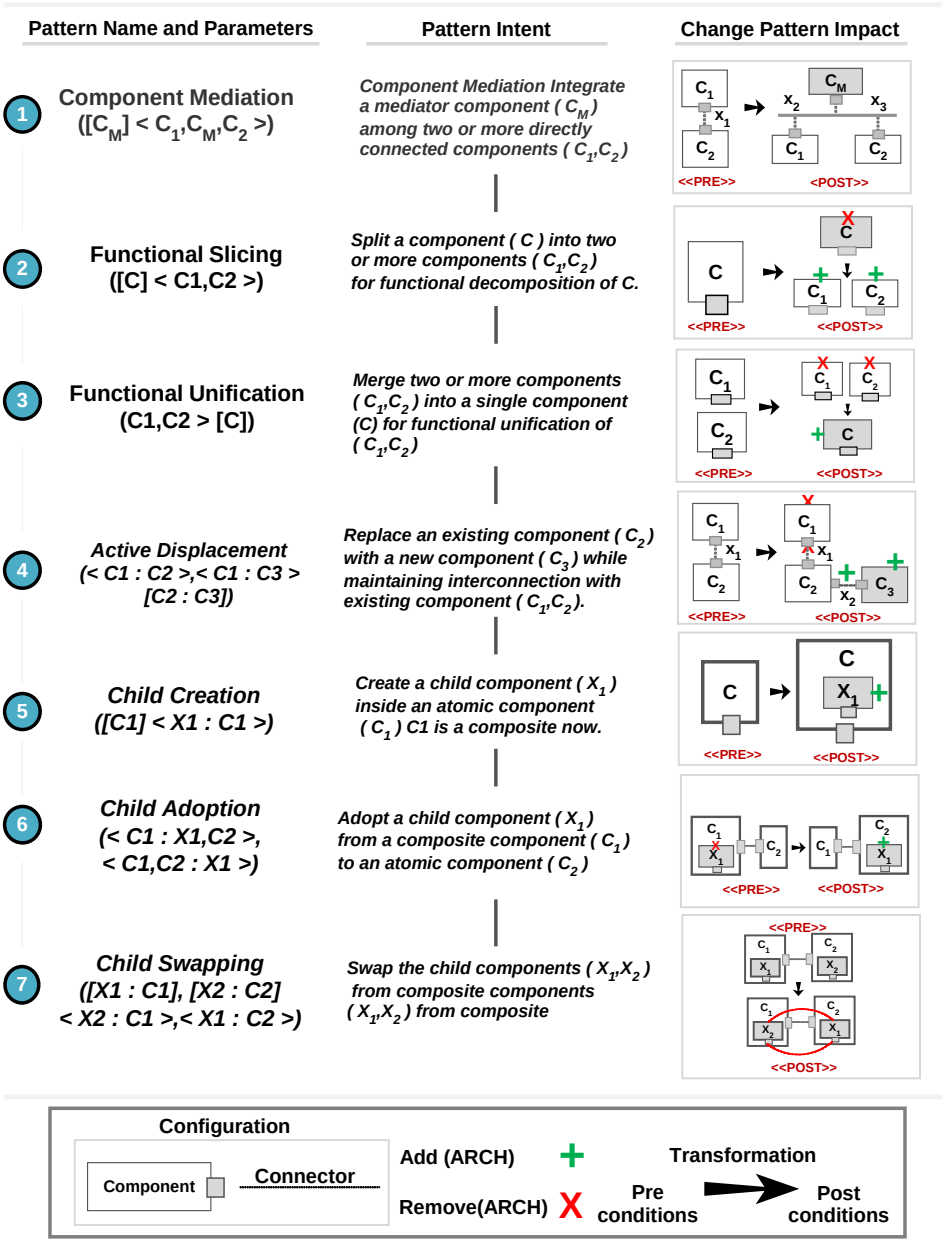


Figure 7. List of Discovered Architecture Change Patterns

5.1 Appplication Domain of Architecture Change Patterns

Patterns are discoverd by investigating change representation in logs of evolving component-based software architecture CBSA models [5,6,25]. Therefore, the applicability (application domain) of the discoverd pattern is limited to CBSA models and their evolution. We have already formally specified CBSA model in Section 2 (cf. Definition 1) - here we only focus on role of change patterns in

supporting reuse of frequent evolution tasks.

5.1.1 Pattern-driven Evolution in CBSAs

We demonstrate pattern-driven reuse by applying discovered pattern instances that support reuse of architectural changes to evolve an existing peer-to-peer (P2P) system to a client server (C-S) architecture - further case study details in [14, 15]. In the existing functional scope two peers components ($P1, P2$) exist such that, $P1$ requests for an appointment to $P2$; whereas $P2$ notifies $P1$ if an appointment is accepted or denied. The existing functional scope of P2P architecture do not support a) multiple requests from P2 to P1 simultaneously and b) registration of an appointment request (by P1) for pending notifications (by P2) if appointment available later on. For illustrative reasons we only discuss following two evolution scenarios (ES):

- **ES-1** - *Integration of a Server Component* that enables handling of multiple requests of appointments by various clients simultaneously. This updates P2P system towards C-S architecture - see Table 5.
- **ES-2** - *Addition of Client Registration and Pending Notification* functionality in Server component. This creates of ClientRegistration and PendingNotification sub-components inside Server - see Table 6.

We claim that a collection of discoverd patterns facilitates an iterative pattern selection and their application to enable an incremental evolution in architectures. By incremental evolution we mean *decomposing evolution process into a manageable set of evolution scenarios that could be addressed in a step-wise manner* [26]. We address scenario ES-1 and ES-2 by applying ComponentMediation and ChildCreation pattern selected from Figure 7. We follow a structured pattern template to address pattern-driven architecture evolution in Table 5. A template-based specification of change patterns provides a structured representation of reusable patterns maintained as change pattern catalogue or a pattern language for architecture evolution.

5.2 Experimental Evaluations of Pattern Discovery Algorithmis

The primary focus of evaluation is to analyse the algorithmic efficiency and accuracy in discovering architecture change patterns from log data.

5.2.1 Evaluation Design

- *Availability of Change Log Data* The evaluation data is gathered by capturing architectural change instances of EBPP and 3-in-1 Phone System case studies (in Section 3) as presented in Table 6. Table 6 presents total change instance as number of change operations on architecture elements. For example, the operation Add on architecture element Component with operational frequency 212 represents addition of 212 architectural components recorded in change log.
- *Scenario-based Analysis of Architecture Evolution* To analyse architectural level evolution, we adopt the architecture level modifiability analysis (ALMA) [13] (detailed in Section 3). To reiterate the concept of change operationalisation on architecture model, we also present two examples in Table 7 **Example I** enables addition of a configuration that follows addition of a component and its port, while **Example II** illustrates removal of endpoint between target and source ports along with removal of connector between CustPayment and BillerCRM components.
- *Algorithmic Efficiency and Accuracy of Pattern Discovery* We aim to investigate:

Table 5. Template-based Specification of Pattern 1 - Component Mediation Pattern

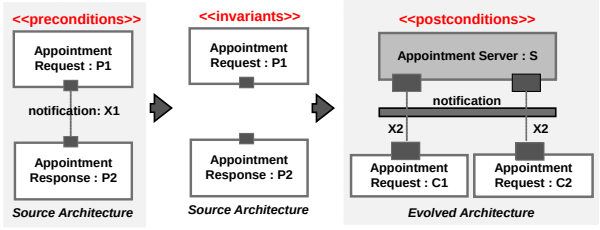
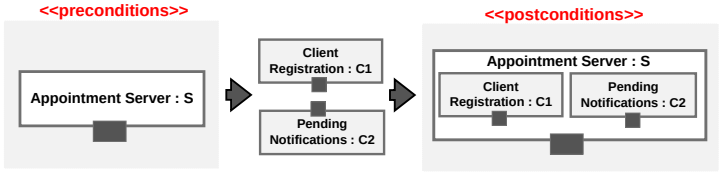
Pattern Description	
Name, Intent, Constraints	**Architecture Change Operations**
Name - ComponentMediation($[S] < P_1, S, P_2 >$) **Intent** - 'Integration of a mediator Server (S) among peer components (P_1, P_2) **Constraints** -Pattern Specific Conditions *Preconditions* - P_1 and P_2 connected using X_1 - **PRE**: $\exists X_1 < (P_1, P_2) \in CMP) \in CON >$ *Invariants* - X_1 among P_1 and P_2 be disconnected. - **INV**: $\nexists X_1 < P_1, P_2) \in CMP) \in CON >$ *Post-conditions* P_1, P_2, S conneced using X_2, X_3. - **POST**: $\exists X_2 < (P_1, S) \in CMP) \in CON >$ $\wedge X_3 < (S, P_2) \in CMP) \in CON >$	**Change Implementation** on architecture model. A. **Add** a Server Component S: $Add(S \in CMP)$ B. **Remove** a P_1 to P_2 Connector X_1: $Rem(X_1(P_1, P_2) \in CMP \in CON)$ C. **Add** a Connector from P_1 to S as X_2: $Add(X_2(P_1, S) \in CMP \in CON)$ D. **Add** a Connector from P_2 to S as X_3: $Add(X_3(P_2, S) \in CMP \in CON)$
Impact of Pattern on Architecture Model	
	
Known Uses - Update a peer-to-peer configuration by integrating a server component among peers. - Add an authentication component between customer billing and payment components, etc.	

Table 6. Template-based Specification of Pattern 2 - Child Creation Pattern

Pattern Description	
Name, Intent, Constraints	**Architecture Change Operations**
Name - Child Creation ($[C_1, C_2] < X(C_1, C_2) >$) **Intent** - To create child components (C_1, C_2) inside an atomic component X **Constraints** on architecture model. PRE: X is an atomic component (Server) INV: Create C_1: ClientRegistration, C_2: Pending Notifications POST: C_1, C_2 created inside X	**Change Implementation** on architecture model. 1. Add a Component C_1: ClientRegistration 2. Add a Component C_2: PendingNotifications 3. Move C_1, C_2 inside X : Server
Impact of Pattern on Architecture Model	
	

1 *Does graph-based representation of change log data enables an efficient searching and retrieval of architectural change instances?*

2 *What is an effective measure of accuracy for pattern discovery process? and how user-driven customisation (through parameterisation) affects pattern discovery process?*

Table 7. Total Change Operations on Architecture Model Recorded from Change Log

	Configuration	Component	Port	Connector	Endpoint	Total
Add	87	212	283	254	297	1133
Remove	36	122	177	163	223	721
Modify	17	52	79	83	113	344
Total	140	386	539	500	633	2198
Example I - Add a Configuration, Component, Port						
Opr_i: = Add(Payment ∈ CFG) Opr_j: = Add(PaymentType ∈ CMP Payment ∈ CFG) Opr_k: = Add(payBill ∈ POR PaymentType ∈ CFG)						
Example II - Remove a Connector, Endpoint						
Opr_m: = Rem(custBill(srcPort, trgPort) ∈ EPT, makePayment ∈ CON) Opr_n: = Rem(makePayment ∈ CMP, (CustPayment, BillerCRM) ∈ CMP)						

5.2.2 Graph Traversal vs File-based Retrieval of Log Data

In Section 3, we already elaborate on how graph-based modeling provides efficient representation of change log data by presenting the intent, scope and operationalisation of changes explicitly. Here we only focus on time-efficiency, i.e time (`T in milli seconds (ms): Y-axis`) to search and retrieve change instances (`per 100 change operations: X-axis`) for traditional file-based retrieval vs graph traversal in Figure 8. The graph in Figure 8 is generated by running 5 different trials to retrieve log data. For example, a typical query to retrieve all instances that represent addition of an architecture element in log is presented as:

$$\text{While } (G_{ACL}.\text{hasNext() != NULL) } \{ \text{ If}(G_{ACL}.\text{getOpr() == 'Add') /* do something */) } \}$$

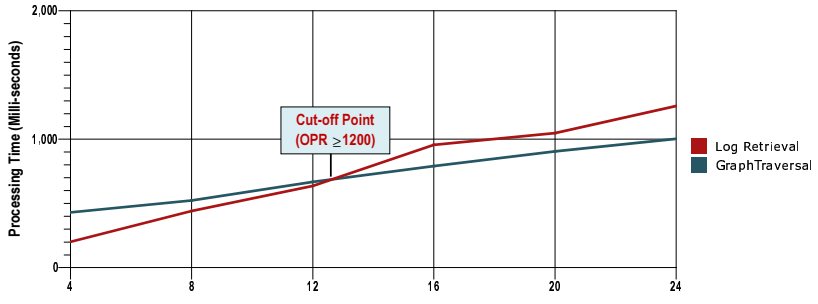
It is vital to mention that to enable graph-based modeling, additional overhead (on average about 400 ms) involves creation of change log graph by mapping change operations and their sequences to graph nodes and edges. However, Figure 8 shows whenever the number of change operation being queried increase by more than 1200 (approximately) - cutoff point - graph traversal and retrieval is always time efficient. We conclude that although graph-based modeling involves additional preprocessing, however we find a cutoff-point at `Retriev(Opr)` $\geq$ `1200 : T(Graph Traversal)` $\leq$ `T(File-based Retrieval)` graph traversal outperforms file-based retrieval.

5.2.3 Algorithmic Accuracy in Discovering Pattern Instances

The objective of this evaluation is of two-fold: i) analyse accuracy of pattern discovery process for mining valid pattern instances and ii) investigate if customisation (user specification of minimum, maximum length of candidates and pattern frequency threshold) enhances accuracy.

While analysing architecture change sequences (cf. Section 4), we observe valid and invalid pattern candidates. More specifically, an invalid candidate sequence (or false positive) represents a recurring change that is a potential pattern in the log, However if we discover such false positives of patterns and use them architectural invariants may be violated. This means to ensure the structural integrity of architecture model, we need to eliminate the invalid candidates. Although, we provide the candidate validation algorithm we also need to ensure accuracy of algorithm in discovering only valid pattern instances. To measure accuracy we use the precision and recall of pattern discovery.

- **Pattern Discovery Precision** - number of *valid pattern instances discovered* $PAT(\varphi)$ divided by the *all pattern (valid and invalid) instances available in log* $PAT(\omega)$, P_{PAT}:= $PAT(\varphi)/PAT(\omega)$



	4	8	12	16	20	24
Log Retrieval	200	440	635	955	1,047	1,258
GraphTraversal	430	522	667	790	905	1,003

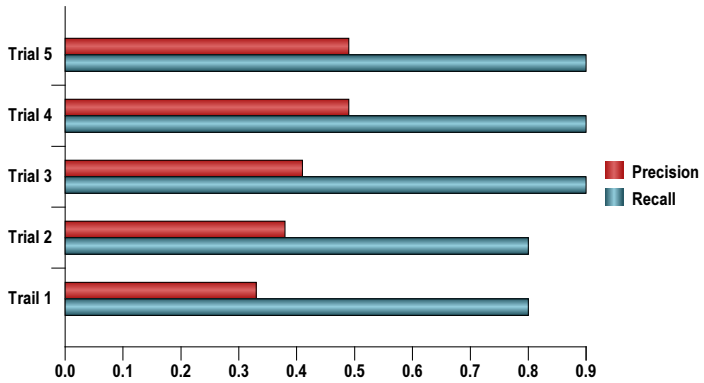
Figure 8. Comparative Analysis of Time Taken (Log-based Retrieval vs Graph-based Traversal) [*(Processor: Intel (R) Pentium (R) 2.16 GHz), (RAM: 2.0 GB), (Windows 7 professional)*]

- **Pattern Discovery Recall** - number of *valid pattern instances discovered* $PAT(\varphi)$ divided by the total *number of valid pattern instances that exist in log* $PAT(\delta)$ $R_{PAT} := PAT(\varphi)/PAT(\delta)$

In Figure 9, we illustrate a relative measure of precision and recall that is based on an average of 5 different trials of pattern discovery over log data. The distinction among trials is maintained by varying the parameters frequency threshold of pattern ($Freq(PAT)$) along with minimum : $minLen(P_C)$ and maximum: $maxLen(P_C)$ pattern candidate length specified by user.

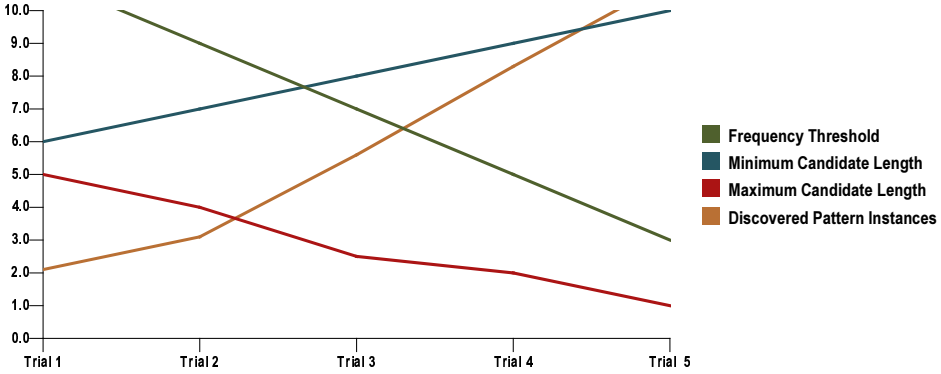
We observe that by applying the pattern candidate algorithm we are able to eliminate the invalid pattern candidates in an incremental fashion. For example, the precision of pattern discovery is 0.33 in first trial, however as we eliminate more invalid instances precision gradually increases and reaches 0.49. At this point, the algorithm is still limited to an accuracy of approximately 0.5 (50 percent) as the probability on average for a valid (0.5) and invalid instance (0.5) remains same. This suggests that candidate validation cannot be fully automated, it required further user intervention to increase pattern discovery precision and to ultimately decrease invalid candidates. In contrast, recall value lies in between 0.9 ± 0.09 corresponding the precision the proposed algorithm is efficient in discovering only valid instances from log (comparison: manual discovery vs automated discovery with algorithm).

In Figure 10, we observe that customising the minimum and maximum length for candidates have a direct impact on total number of discovered pattern. It is evident that minimum length specification for candidate generation is inversely proportional to discovered instances ($minLen(P_C)$ $1/\propto PAT$), i.e when we decrease minimum length (from 5 to 1: such that $1 \leq minLen(P_C) \leq 5$) discovered pattern instances increase from 2.1 to 10.9. On the contrary, maximum length specification for candidate generation is directly proportional to discovered instances $maxLen(P_C) \propto PAT$), i.e when we increase maximum length (from 6 to 10: such that $6 \leq maxLen(P_C) \leq 10$) number of discovered pattern increases. Frequency threshold is inversely proportional to discovered instances ($Freq(C_P)$ $1/\propto PAT$), i.e when we decrease minimum length (from 11 to 3: such that $3 \leq Freq(C_P) \leq 11$) discovered pattern instances increase from 2.1 to 10.9



	Trail 1	Trial 2	Trial 3	Trial 4	Trial 5
Precision	0.33	0.38	0.41	0.49	0.49
Recall	0.8	0.8	0.9	0.9	0.9

Figure 9. Precision and Recall of Pattern Discovery Algorithms (Valid vs Invalid Instances)



	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
Frequency Threshold	11.0	9.0	7.0	5.0	3.0
Minimum Candidate Length	6.0	7.0	8.0	9.0	10.0
Maximum Candidate Length	5.0	4.0	2.5	2.0	1.0
Discovered Pattern Instances	2.1	3.1	5.6	8.3	10.9

Figure 10. Co-relation of Pattern Instances Discovered vs Customisation of Discovery Parameters

6 Related Work

Our research lies at the intersection of two distinct domains, i.e; i) software repository mining [10] and ii) software evolution [2]. More specifically, we focus on *architecture change mining* that aims to exploit repository mining concepts to investigate histories of architecture evolution for discovery of architecture change patterns. To position and justify proposed contribution, we overview state-of-the-research that address: i) Pattern Application - methods and techniques to enable pattern-driven reuse (Section 6.1) and ii) Pattern Acquisition - solution for discovering change patterns (Section 6.2). We also review graph-based techniques that can be extended and/or customised to develop algorithms

that automate and customise the process of discovering architectural change patterns.

6.1 Pattern and Style-based Reuse of Architecture Evolution

In recent years, the emergence of change patterns [8,11] and evolution styles [5,6] promoted solutions that can leverage reuse knowledge and expertise to tackle recurring problems in architecture evolution. Both change patterns and evolution styles although conceptually innovative, they build-upon the more conventional philosophy behind and *design patterns* [27] and *architectural styles* [28] to address evolution-centric issues in software architectures. Change patterns [8] follows-on reuse-driven 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 [29]) to support both the design-time as well as **run-time** evolution. Prominent solution with pattern-based reuse includes co-evolution of processes and requirements [8] to their underlying architecture models as **design-time** evolution. Adaptation and reconfiguration patterns [30] are the only exception for run-time evolution. In contrast to patterns, evolution styles focus on defining, classifying, representing and reusing frequent evolution plans and architecture change expertise. Style-based approaches are limited in addressing corrective and perfective changes implemented as design-time evolution. In the Style-driven approaches, most notable trends include structural Evolution-off-the-Shelf [6] and evolution planning [5] with time, cost and risk analysis to derive evolution plans.

In our systematic literature review to classify and compare state-of-the-research for evolution reuse in ACSE [3, 4] we observed that, although, evolution styles and change patterns promote application of evolution-centric knowledge in architecture evolution; there is a clear lack of focus on acquisition of reuse knowledge that involves a continuous discovery of new styles and patterns. Moreover, to enhance or enable an anticipated and reuse-driven architecture change management there is a growing need of solutions that facilitate a continuous discovery of reuse knowledge - including frameworks, processes and patterns etc. - by investigating evolution histories [10,11]. The state of existing research suggests the needs to extend solutions that support architecture change mining (knowledge acquisition) as a complementary and integrated phase to architecture change execution (knowledge application).

6.2 Discovery of Architecture Change Patterns

In contrast to pattern application to drive architecture evolution (cf. Section 6.1), research on discovery of architecture change patterns is not well established. Based on a systematic review [3], the only notable work refers to identification of architectural change patterns from object-oriented software [31]. The fundamental distinction between the two approaches is the level of discovery that represents different software abstraction in terms of pattern discovery from source codes changes and architecture evolution.

- *Level of Discovery* refers to the software abstraction that is subject to investigation for pattern discovery. The objective of our comparison is to focus on investigation of specific abstraction level of software for effective change management. In [31] the authors focus on pattern identification from software source code changes, while we propose pattern discovery by analysing architecture-centric evolution.
- *Source of Discovery* refers to an established repository infrastructure providing fine-grained instances of change for pattern discovery. By comparison, we highlight the role of software change repositories and exploiting evolution-centric data contained in them to study evolution. In contrast to mining version control systems we investigate architecture change logs to perform post-mortem analysis of architecture evolution history with special focus on pattern discovery.

- *Pattern Discovery Method* refers to methods and techniques for pattern discovery. The comparison promotes application of available methods and that can be extended or enhanced in future research for pattern discovery. The solution in [31] compares two versions of source code changes to identify patterns of change from one version to another. We propose to exploit sub-graph mining - to discover recurring sub-graphs in architecture change graphs - as discovered patterns.
- *Tool Support* refers to available prototype with primary intent to support automation and customisation of pattern discovery process. The objective of comparison is to promote tools and automation to analyse evolution histories vital when evolution data is complex and of significant size to restrict manualy analysis. In [31] prototype toolkit HEAT (High-level Evolution Analysis) is developed to implement and evaluate the approach. We provide a prototype G-Pride (Graph-based Pattern Identification) for automation and proof-of-concept purposes.
- *Formal Support of Discovery* refers to application of formal methodologies to discover patterns. By comparison, we aim to highlight the role of formal methodologies with an inherent support for formal evaluation of discovery process. In contrast to comparing the code version snap shots [31], we apply graph isomorphism (recurring changes sequences in architecture change graphs) to discover patterns.

6.3 Graph-based Pattern Discovery

The concept of discovering sequential patterns was first presented in [32]. Since then, there is a growing development of algorithms and mathematical proof for mining sequential patterns across different domains [19, 20]. The solution to our pattern identification problem lies within Frequent Subgraph Mining (FSM) techniques [19], with a comprehensive state-of-the-art detailed in [20]. More specifically, within FSM we focus on apriori-based approach that proceeds in a generate-and-test manner using a Breadth First Search (BFS) strategy during each iteration to (i) generate and validate pattern candidates (ii) to determine the occurrence frequency of candidates in the architecture change graph. In our solution, during the candidate generation once-off monotonic constraints (constrained-based mining [33]) can be enforced to minimise the generation of invalid candidates. We achieve this using candidate validation process that removes the false positives i.e. candidates that are potential patterns but their application violates structural integrity of architecture elements.

It is also worth mentioning about Graph X-Ray (GRay) [19], that works on attributed graphs to find sub-graphs that either match the desirable query pattern exactly, or as close as possible based on pre-defined criteria. We are specifically concerned about identifying patterns in large to medium attributed graphs where graph nodes and edges have multiple attributes that contain instances of architecture elements and pattern-specific constraints. In such circumstances, we may want to identify such change patterns that Integrates a new component among two or more existing components or Replace an existing component with a new one. In addition to an exact pattern match (sub-graph isomorphism) we also focus on identifying in-exact instances of a pattern (exact and in-exact match detailed in Section 4).

References

1. T Mens SD (2008) Software Evolution, Springer. First edition.
2. Lehman M (1996) Laws of software evolution revisited. In: Software Process Technology. Lecture Notes in Computer Science.

3. A Ahmad PJ, CPahl (2003) Classification and comparison of evolution reuse knolwedge in software architectures - a systematic review. Technical Report, School of Computing, Dublin City University.
4. P Jamshidi AA M Ghaffari, Pahl C (2012) A framework for classifying and comparing architecture centric software evolution. In: In 17th European Conference on Software Maintenanace and Reengineering. IEEE Computer Society.
5. J Barnes DG, Schmerl B (2012) Evolution styles: Foundations and models for software architecture evolution. Journal of Software and Systems Modeling .
6. O Le Goaer MO D Tamzalit, Seriai A (2008) Evolution shelf: Reusing evolution expertise within component-based software architectures. In: In International Conference of Computer Software and Applications. IEEE Computer Society.
7. H P Breivold IC, Larsson M (2012) A systematic review of software architecture evolution research. Information and Software Technology 54: 16–40.
8. K Yskout RS, Joosen W (2012) Change patterns: Co-evolving requirements and architecture. In Journal of Software and Systems Modeling .
9. Ahmad A, Pahl C (2012) Design patterns: Abstraction and reuse of object-oriented design. In: ERCIM News, 88.
10. H Kagdi MLC, Maletic J (2007) A survey and taxonomy of approaches for mining software repositories in the context of software evolution. Journal of Software Maintenance .
11. A Ahmad PJ, Pahl C (2012) Graph-based pattern identification from architecture change logs. In: In International Workshop on System/Software Architectures. Lecture Notes in Computer Science.
12. Liguo Y (2009) Mining change logs and release notes to understand software maintenance and evolution. In CLEI Electronic Journal .
13. P Bengtsson JB N Lassing, Vliet HV (2004) Architecture-level modifiability analysis. Journal of Systems and Software 69.
14. The electronic payments association (nacha). [Online]: http://www.nacha.org. 59, 161, 169.
15. (1999) Bluetooth consortium, k3: Cordless telephony profile, bluetooth specification. Version 1.0.
16. A Ahmad MA P Jamshidi, Pahl C (2012) Graph-based implicit knowledge discovery from architecture change logs. In: In 7th Workshop on SHAring and Reusing architectural Knowledge.
17. H Ehrig UP, Taentzer G Fundamental theory for typed attributed graph transformation. In: In Graph Transformations.
18. U Brandes IHMH M Eiglsperger, Marshall MS (2002) Graphml progress report structural layer proposal. In: In Graph Drawing.
19. H Tong BG C Faloutsos, Eliassi-Rad T (2007) Fast best-effort pattern matching in large attributed graphs. In: 13th ACM International Conference on Knowledge Discovery and Data Mining.
20. C Jiang FC Chuntao, Zito M (2012) A survey of frequent subgraph mining algorithms. Knowledge Engineering Review .
21. N PA Harrison, Zdun U (2007) Using patterns to capture architectural decisions. In IEEE Software .
22. Buckley MZAR T Mens, Kniesel G (2005) Towards a taxonomy of software change. journal of software maintenance and evolution. Journal of Software Maintenance and Evolution .

23. U Brandes DW M Gaertler (2007) Experiments on graph clustering algorithms. In: 11th Annual European Symposium. Lecture Notes in Computer Science.

24. A Ahmad PJ, Pahl C (2013) A pattern language for evolution support in component-based software architectures. In: Technical Report - School of Computing, Dublin City University, Ireland. www.computing.dcu.ieãaakash.pdf.

25. N Medvidovic DR, Taylor RN (1999) A language and environment for architecture-based software development and evolution. In: International Conference on Software Engineering. IEEE Computer Society.

26. Goedicke M, Zdun U (2002) Piecemeal legacy migrating with an architectural pattern language: A case study. Journal of Software Maintenance and Evolution: Research and Practice 14: 1–30.

27. E Gamma RJ R Helm, Vlissides J (1993) Design patterns: Abstraction and reuse of object-oriented design. In: In Object-Oriented Programming (ECOOP).

28. C Pahl SG Claus, Hasselbring W (2009) Ontology-based modelling of architectural styles. In Information and Software Technology .

29. Williams B, Carver J Characterizing software architecture changes: A systematic review. Information and Software Technology 52: 31–51.

30. H Gomaa MKSM K Hashimoto, Menasc D (2010) Software adaptation patterns for service-oriented architectures. In: Proceedings of the 2010 ACM Symposium on Applied Computing.

31. Dong X, Godfrey M (2008) Identifying architectural change patterns in object-oriented systems. In: In 16th IEEE International Conference on Program Comprehension. IEEE Computer Society.

32. R Agrawal RS Mining sequential patterns. In: In International Conference on Data Engineering.

33. Ullmann JR (1976) An algorithm for subgraph isomorphism. Journal of the ACM 23: 31–42.