



Proceedings of the
First Workshop on Patterns Promotion
and Anti-patterns Prevention
(PPAP 2013)

A Pattern Language for the Evolution of Component-based
Software Architectures

Aakash Ahmad, Pooyan Jamshidi, Claus Pahl and Fawad Khaliq

31 Pages

A Pattern Language for the Evolution of Component-based Software Architectures

Aakash Ahmad, Pooyan Jamshidi, Claus Pahl and Fawad Khaliq[†]

Lero – the Irish Software Engineering Research Centre
School of Computing, Dublin City University, Ireland

[†]Department of Computer Science, Quaid-i-Azam University, Islamabad, Pakistan
[ahmad.aakash || pooyan.jamshidi || claus.pahl]@computing.dcu.ie,

[†]fawad.khaliq@cs.qau.edu.pk

Abstract: Architecture-centric software evolution enables change in a system’s structure and behaviour while maintaining a global view of the software to address evolution-centric trade-offs. The existing solutions for architectural maintenance and evolution fall short of exploiting generic and reusable expertise to address recurring evolution problems. We present a pattern language as a collection of interconnected change patterns that enable reuse-driven and consistent evolution of component-based software architectures. Pattern interconnections represent possible relationships among patterns (such as variants or related patterns) in the language. In general, we introduce architecture *change mining* (pattern language development) as a complementary and integrated phase to facilitate reuse-driven architecture *change execution* (pattern language application). We evaluate the language applicability to support pattern-driven reuse in architecture evolution of a payment system case study. We also analyse the precision and recall factor as a measure of selecting the most appropriate pattern(s) from the language collection. The pattern language itself continuously evolves with an incremental discovery of new patterns from change logs over time.

Keywords: Pattern Definition, Pattern Detection, Pattern Language, Architecture Evolution

1 Introduction

Modern software systems operate in a dynamic environment with frequent changes in stakeholders’ needs, business and technical requirements, and operating environments [MS08]. These changing requirements trigger a continuous evolution in deployed software that must be addressed while maintaining a global view of the system to effectively resolve evolution-centric trade-offs [Leh96]. Component-based software architecture (CBSA) [MRT99] [BGS12] models a system’s structure as hierarchical configurations of computational components and their interconnections by abstracting implementation-specific details. The role of CBSA – as a blue-print of evolving software – is pivotal to fill the gap between changing requirements [YSJ08] and refactored source code [BGS12] [GTO⁺08] [TM08].

Lehman’s law of continuing change [Leh96] poses a direct challenge to research and practices that aim to support long-living and continuously evolving architectures [GTO⁺08] under frequently varying requirements [YSJ08]. The law states that “...systems must be continually adapted or they become progressively less satisfactory”. The implications of a continuous change have resulted in development of solutions that enable reusable, off-the-shelf expertise to address recurring evolution problems. The existing solutions promoted the ‘build-once, use-

often' philosophy by exploiting change patterns [YSJ08] [CHW07] and evolution styles [GTO⁺08] [TM08] to address a continuous evolution of software architectures. In contrast to pattern invention [YSJ08] [TM08], we propose a continuous (a) discovery of new patterns, their (b) specification and reuse to (c) apply pattern-based evolution. In our systematic review [AJP12a], we observed that there is a need for solutions – supporting a continuous acquisition and application of architecture evolution reuse knowledge – to address recurring problems (i.e., frequent restructuring of architectural model) during evolution process [JGA⁺12].

We propose a language (PatEvol) as a formalism and collection of (empirically discovered) architecture change patterns [AJP12b] that provide reusable solutions to recurring evolution problems. Pattern interconnections allow possible relationships to exist among patterns to enable the composition of a pattern language [GZ02] [Zdu07]. By exploiting the vocabulary and grammar of a language, individual patterns can be formalised and interconnected to support reusable and off-the-shelf architectural evolution. Our solution is fundamentally inspired by Alexander's seminal theory [Ale99] about pattern languages that integrate patterns as repeatable solutions to build architectures. We identify the central research challenge as:

How to utilise architectural evolution knowledge that allows us to model and execute reusable changes to evolve component-based software architectures?

We highlight the challenges and central features of pattern-based evolution as:

A. Pattern-based Reuse in Architecture Evolution – Reuse-knowledge in the proposed pattern language is expressed as a formalised collection of interconnected patterns (a.k.a pattern relations) [Zdu07]. The proposed pattern language is composed of (7+2) patterns; representing seven newly discovered architecture change patterns and two variants for one of the patterns. Architecture change patterns abstract the primitive changes (addition, removal, modification, etc. of components and connectors) into reusable pattern-based changes (composition, decomposition, replacement, etc. of components and connectors).

B. Pattern Selection Problem – The pattern selection problem is a significant challenge for inexperienced developers or architects to search and select the appropriate patterns from large collections [KZ07]. With a language-based formalism, we exploit the Question-Option-Criteria (QOC) methodology [MYB⁺91] to address the pattern selection problem. The complexity of the pattern selection problem increases when new patterns are discovered [AJP12b] and integrated in the pattern language. The QOC methodology enables the selection of appropriate pattern(s) by evaluating their impacts and consequences [CHW07] [Zdu07].

C. Application Domain of the Pattern Language – Component-based architecture models [MRT99] [HTW⁺99] and their evolution define the application domain of our pattern language. Patterns in the language enable composition, decomposition, or replacement of components and their interconnections in the architecture. The proposed contributions are:

- This paper offers a significant extension of previous research on architecture change pattern discovery [AJP12b] [AJA⁺12] with a pattern language as a system-of-patterns for architecture evolution.
- In contrast to pattern [TM08] and style [BGS12] invention, we discover patterns [AJP12b] and specify them in a collection to enable their future reuse.
- In general [JGA⁺12] [AJP12a], we unify the processes of i) architecture change mining to discover architecture evolution knowledge that guides ii) architecture change execution.

The rest of this paper is organised as follows. Related research is discussed in Section 2. Background on pattern discovery and pattern modeling is detailed in Section 3. Research challenges and proposed solutions are presented in Section 4. Details about the composition of a pattern language are provided in Section 5. The application domain of the pattern language is presented in Section 6. We illustrate the application of pattern language to evolve software architectures in Section 7. Implementation of a prototype and evaluation of the solution are presented in Section 8. Conclusions and future work are outlined in Section 9.

2 State-of-Research on Patterns for Architecture Evolution

In the software architecture community, *pattern oriented software architecture* [BHS07] represents one of the foundational literature on patterns and pattern languages for architecture design. In contrast to patterns of *architectural design* [BHS07], our solution is the first attempt towards promoting a pattern language to enable reuse in *architectural evolution*.

We conducted two systematic literature reviews (SLRs) to classify and compare the existing research that supports acquisition and application of architecture evolution-reuse knowledge to guide architectural maintenance and evolution [JGA⁺12] [AJP12a]. In [AJP12a], we define architectural 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*”.

Based on the results of these SLRs, in Table 1 we highlight and compare the core features of the relevant existing research with our solution. Table 1 provides the basis for a comparative analysis (potential and limitations) of our solutions in the context of existing research. The growing research needs for pattern languages [BHS07] are also highlighted with a dedicated series of conferences such as PLoP¹ and EuroPLoP². We are specifically interested to present:

- (a) *What* are the existing approaches that enable reuse-driven evolution in architectures? and *How* are the proposed solutions identical or unique to the existing ones? (See Section 2.1)
- (b) *What* is the role of pattern languages in supporting architecture change management? and *Why* is there a need for pattern language(s) to evolve software architectures? (See Section 2.2)

2.1 Reuse-Driven Evolution in Software Architecture

In the context of architecture evolution-reuse knowledge [AJP12a], *evolution styles* [BGS12] [GTO⁺08] [TM08], and *change patterns* [YSJ08] [CHW07] [GZ02] emerged as the only notable solutions to enable reuse of design-time as well as runtime evolution of architectures. Evolution styles and change patterns build on the conventional concepts of architecture styles and change patterns to address architectural evolution.

– **Evolution Styles** it is interesting to observe that research in [BGS12] [GTO⁺08] exploits the same concept (i.e., evolution styles) but address two distinct problems in architecture evolution. More specifically, [BGS12] is the pioneering work on style-driven evolution and is focused on defining, classifying and reusing frequent evolution plans [BGS12]. In contrast to

¹PLoP - Conference on Pattern Languages of Programs: <http://www.hillside.net/plop>

²EuroPLoP - European Conference on Pattern Languages of Programs: <http://www.europlop.net/>

the solution in [BGS12], the authors in [GTO⁺08] exploit styles for reusable architecture refactoring. The existing research lacks a consensus about what exactly defines an evolution style, and what is a precise role of evolution styles in architectural change management. In style-driven approaches, notable trends are structural evolution-off-the-shelf [BGS12] and evolution planning [BGS12] with time, cost, and risk analysis to derive evolution plans.

Table 1. A Comparison Summary of Proposed Solution (PatEvol) with State-of-the-Research

Reuse Method	Proposed Solutions	Type of Reuse	Time of Evolution	Architecture Descriptions	Tool Support
Style-based Reuse	Evolution Styles [BGS12] [GTO ⁺ 08]	Evolution Plans	Design-time	Component and Connector	AEvol
Pattern-based Reuse	Change Patterns [YSJ08] [CHW07]	Co-Evolution Patterns	Design-time & Runtime	Component and Connector	VIATRA
Pattern Languages	Pattern Languages [GZ02] [HZ06]	Migration Patterns	Design-time	Object and Service-Oriented	MDS Tool Chain
Proposed Solution	Change Pattern Language	Change Patterns & Operators	Design-time	Component and Connector	AEvol

– **Change Patterns** follow reuse-driven methods and techniques to offer a generic solution to frequent evolution problems. In [JGA⁺12], we observe that existing solutions overlook the needs for an empirical discovery of evolution patterns. In our systematic reviews of architecture evolution [AJP12a], we observed that architecture change patterns [YSJ08] [CHW07] are mostly a proposed solution (based on individual experience) that undermines the fact that patterns represents reuse knowledge that must be empirically discovered.

Based on the comparison in Table 1, our solution is fundamentally similar to [GTO⁺08] in terms of enabling evolution reuse. However a most notable difference is that instead of inventing styles as architecture evolution reuse knowledge, we propose an empirical discovery of change patterns [AJP12b]. During architectural evolution, we also support a (semi-) automated selection of the appropriate change pattern(s) from a collection of the patterns in the language. Considering architecture evolution process, we support a two-step solution for a continuous discovery and application of patterns. In contrast to adaptation or reconfiguration patterns [YSJ08] [GH04] that support runtime evolution, our solution is limited to supporting design-time evolution.

2.2 Pattern Languages for Architecture Change Management

Pattern languages provide a formal grammar, vocabulary, and pattern sequencing to derive structure and semantic relationships of patterns in a collection. In the context of architectural change management, the only notable research is on legacy migration [GZ02] and process-oriented integration [HZ06] of software architectures. In [GZ02], the authors propose an incremental migration of legacy software to a flexible architecture using migration patterns. This solution offers a pattern language for migrating C language implementations to components in an object-oriented system.

Based on a comparison in Table 1, in contrasts to [GZ02] our solution is not focused on migration of legacy code to components, instead it supports reuse of architecture evolution. We propose that change patterns [AJP12b] [AJA⁺12] as generic reusable abstractions must be empirically *identified as recurring, specified once, and instantiated multiple times* to benefit evolving architectures.

3 Change Pattern Discovery and Modeling

In this section, we explain the aspects of *pattern discovery* and *pattern modeling* as background details before presenting the pattern language.

3.1 Patterns Discovery from Architecture Change Logs

Based on an existing evidence of pattern-based evolution [YSJ08] [CHW07], we observed a lack of experimental methodologies for pattern discovery. In contrast to existing solutions [CHW07], we propose change patterns as discovered knowledge – from established knowledge sources – that can be shared and reused to guide evolution. Our solution for change pattern discovery is conceptually similar to [BGA06] that is focus on extracting change patterns from CVS repositories. However, in contrast to mining patterns for co-changing groups of source code files, we discover architecture change patterns from architecture change logs [AJP12b]. A *change log* represents a transparent and centrally manageable repository of change operations on architecture model [AJA⁺12]. By investigating change logs, we perform the post-mortem analysis of architecture evolution to discover usage-determined change patterns. We summarise the essential aspects of pattern discovery below. Additional details about our algorithmic solution, formalism and tool support for pattern discovery are presented in [AJP12b]. A technical report with extended details of pattern discovery from architecture change logs is provided in [AJP12d].

- *Source of Pattern Discovery* is represented as architecture change logs [AJA⁺12] that maintain traces of architecture *change history*. Log-based investigation of architecture evolution provides an updated central repository with analytical support to search, query, and analyse change representation and to ultimately discover change patterns.
- *Formal Methodology for Pattern Discovery* is supported by means of graph-based models to represent change instances from logs as an attributed graph [EPT04]. Graph-based modeling of architecture change allows us to utilise the *sub-graph mining* approaches [AJP12b] to (a) analyse change operationalisation, (b) identify operational dependencies and (c) to discover recurring change s patterns.
- *Algorithmic and Tool Support for Pattern Discovery* we introduce the pattern discovery problem as a modular solution [AJA⁺12]. It enables automation along with appropriate user intervention and customisation through parameterisation for the pattern discovery process [AJP12b]. We provide a prototype (G-Pride: Graph-based Pattern Identification) for automation and scalability of the pattern discovery process.

3.2 Meta-model of Architecture Change Patterns

In architecture change logs [AJA⁺12], we observed that architectural changes can be operationalised and parameterised to support architecture evolution. More specifically, architecture elements that are added, removed, or modified are specified as parameters of change operations. This operationalisation of architectural changes helps us to define architecture 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 as Replace (C1, C2). We express pattern-based evolution as: PatEvol:= <ARCH, OPR, CNS, PAT, COL>. In

Figure 1, the meta-model provides a structural composition of the pattern elements and allows a formal specification of pattern language grammar [Zdu07] (in Section 4).

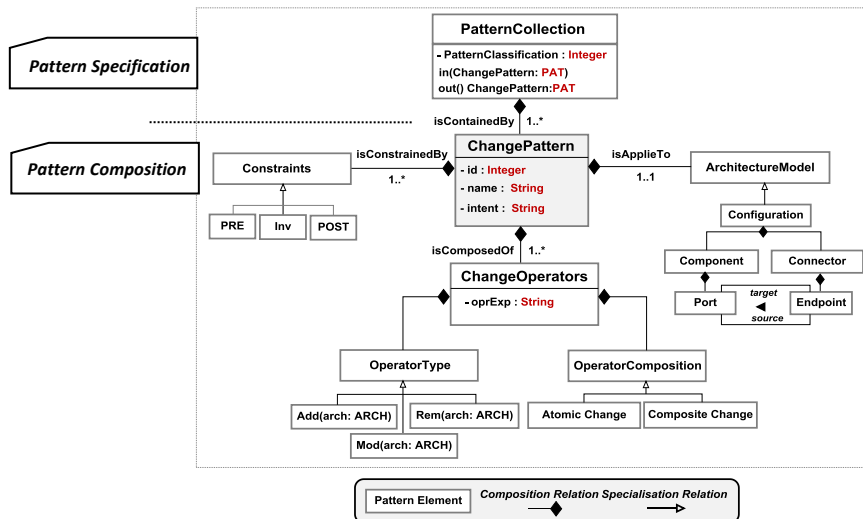


Figure 1. A Meta-model for Change Pattern Composition.

1. **Architecture Model (ARCH)** refers to the architecture elements to which a pattern can be applied during change execution. We represent the CBSA model as topological configurations (CFG) based on a set of architectural *components* (CMP) as the computational entities, linked through *connectors* (CON) [MRT99]. Furthermore, architectural components are composed of component *ports* (POR), and connectors are composed of *endpoints* (EPT) to bind component ports. The consistency of pattern-based change and structural integrity of architecture elements beyond **component-based** (also **service component**) model is undefined.

Change patterns in this paper address component-based software engineering in general and existing research on component-based software architecture and their evolution [BGS12] [GTO⁺08] in particular. We believe that architecture descriptions in the meta-model can be extended to model more conventional object-oriented architectures [GZ02]; however this possibility can only be seen as a future work. Additional details about component-connector architectures along with the evolution scenarios are provided in Section 6.

2. **Change Operators (OPR)** represents change instances that are fundamental to operationalising architectural evolution. Our analysis of the change log [AJP12b], [AJA⁺12] goes beyond basic change types to specify a set of atomic and composite operations enabling structural evolution by adding (ADD), removing (REM), and modifying (MOD) elements in CBSAs. 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, addition of a new port P in an existing component C is expressed as follows (ϵ represents type of element): Add ($P \in \text{POR}, C \in \text{CMP}$).

– *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 ($<$ represents operational sequence).

$$\text{Add}(C \in \text{CMP}, G \in \text{CFG}) < \text{Add}(P \in \text{POR}, C \in \text{CMP})$$

Change operators represent *primitive changes* [BGS12] [AJA⁺12] that are composed into *pattern-based changes* [CHW07] [AJP12b] – abstracting addition, removal, and modification of components and connectors with composition, decomposition, and replacement type changes in an architecture model.

3. Constraints (CNS) 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 post-conditions ensure that a component C containing a port P has been successfully added in a configuration G .

4. Change Patterns (PAT) defines a first-class abstraction that can be operationalised and parameterised to support potentially reusable architectural change execution expressed as:

$$\text{PAT}\langle \text{name}, \text{intent} \rangle: \text{PRE}(ae_m \in \text{ARCH}) \xrightarrow{\text{INV}(\text{OPRn}(ae_m \in \text{ARCH}))} \text{POST}(ae'_m \in \text{ARCH})$$

A pattern has a name and an intent that represents a recurring, constrained (CNS) composition of change operationalisation (OPR) on architecture elements ($ae_m \in \text{ARCH}$) – in Figure 1. We further discuss discovered pattern and their variants in Section 5.

5. Collection (COL) 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. Pattern specification, selection, and retrieval are detailed in Section 7.

The background details about pattern discovery and representation help us to outline the research challenges and proposed solution (in Section 4) with structural composition of change patterns language (in Section 5).

4 Research Challenges and Proposed Solution

In Section 3 (cf. Figure 1), an individual change pattern outlines the core of a (repeatable) solutions to resolve recurring evolution problems. The potential of change patterns can only be achieved if individual patterns are applied in the context of each other – establishing pattern relations – known as a *system or language of patterns* [GZ02] [Ale99] [Zdu07]. We propose that, language-based collection facilitates an iterative pattern selection and their application to enable an incremental evolution. By incremental evolution, we mean decomposing the *evolution process into a manageable set of evolution scenarios that could be addressed in a step-wise manner* [GZ02]. We identify the challenges in Section 4.1 and present our solution in Section 4.2.

4.1 Research Challenges

Based on change pattern discovery (cf. Section 3.1), we outline the core research challenges as: *i) establishing* pattern relationships in a language context, and *ii) selecting* appropriate patterns in a given evolution scenario. Details about pattern discovery are already presented in Section 3.1 (our previous research [AJP12b], [AJA⁺12], [AJP12d]). We now focus on:

Challenge I – Development of Pattern Relationships refers to the possible interconnection(s) among patterns in a collection (language) that build on each other to support an incremental solution (step-wise pattern application) for evolution problems.

The challenge lies with establishing the relationships or an order of application for individual patterns in the language collection [GZ02] [Zdu07]. Pattern interconnection requires creation of either *static*, *dynamic*, or both types of relations among change patterns. Static or predefined relations express specialised and generalised type of patterns in the language. Static pattern relations are limiting when expressing sequential relations among the patterns in the language. In contrast, sequential or *dynamic* relations determine if a pattern is dependent-on or independent-of other patterns. We further discuss pattern relations in Section 5.

Challenge II – Selection of Patterns from a Collection refers to a flexible mechanism for querying, searching, and selecting the most appropriate pattern from a language collection.

Pattern selection is a significant challenge for inexperienced developers or architects in terms of searching required patterns in continuously-updated collections and selecting the appropriate patterns or possible alternatives [KZ07]. Systematically selecting the appropriate patterns requires a certain amount of expertise from the software architect or the designer. More specifically, the architect/designer must understand how a pattern solution fits into the overall architecture evolution problem and how other patterns can be applied to resolve consequences of applying the first pattern. Some of the typical questions arising for an architect could be: (1) *Which pattern should I choose first?*, (2) *Which variant of the pattern works best?*, and (3) *Which pattern should be applied next?* We further discuss pattern selection and pattern application in Section 7.

4.2 Overview of the Proposed Solution

To support a formalised system-of-patterns for reusable evolution and addressing the challenges in Section 4.1, we illustrate our solution in Figure 2. We propose *architecture change mining* as a complementary and integrated phase to *architecture change execution*. Therefore, the reuse knowledge and expertise discovered during change mining can be shared and reused (as patterns) to guide architecture change execution.

A. Architecture Change Mining – we propose *architecture change mining* process (language composition) by exploiting the software repository mining concepts [KCM07] [BGA06] to investigate architecture change logs [AJA⁺12]. The ultimate outcome of architecture change mining process is pattern discovery and composition of change pattern language. The language presents a formalised collection of change patterns to map problem-solution view of the domain (i.e., evolving CBSAs). The discovered patterns and their variants represent the *language vocabulary* as in Figure 2. To express the structural composition of pattern (meta-model) that governs the relations among pattern elements (i.e., *language grammar*).

– *Addressing Challenge I – Development of Pattern Relationships*: in our solution an important task includes creating an interconnection-of-patterns that provides a foundation to establishing the relations among change patterns (i.e., *pattern sequencing*). Reuse-knowledge is expressed as a collection of patterns to enable a generic and reusable evolution in CBSAs.

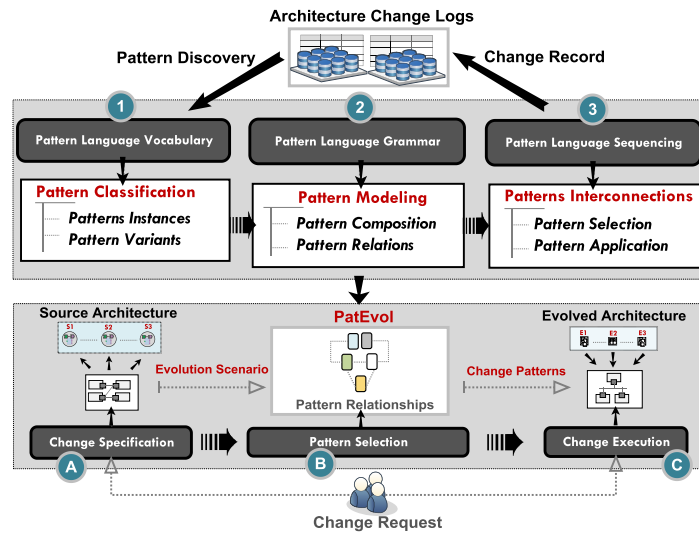


Figure 2. Overview of Proposed Solution – Change Mining (PatEvol Development) and Change Execution (PatEvol Application).

B. Architecture Change Execution – we propose architecture *change execution* process (language application) by exploiting software evolution concept [MS08] [Leh96] to enable pattern-driven reuse in architecture evolution. In Figure 2, we propose architecture *change execution* that enables pattern-driven reusable evolution of component-architectures. In the proposed solution, an architect specifies change request (as addition, removal, or modification) of architectural elements in existing CBSA. A declarative specification of change requests enables the selection of appropriate pattern sequences to derive a pattern-driven evolution strategy based on given evolution scenarios.

– *Addressing Challenge II – Pattern Selection Problem*: during change execution, the pattern selection problem is addressed by adopting and customising the Question-Option-Criteria (QOC) methodology [MYB⁺91]. The QOC methodology helps us to select the most appropriate pattern from the language collection by evaluating the forces and consequences of a given patterns during evolution [YSJ08] [CHW07] [GZ02]. Currently, the pattern language is composed of 7 change patterns and 2 variants of one of the discovered pattern. Although the current (7+2) patterns do not represent an extensive collection, the pattern selection problem increases as the number of newly discovered patterns in the language grows – how a newly discovered pattern can be incorporated in a language grammar? (see Section 5).

5. Composition of Change Pattern Language

The pattern language is formally composed of (a) a classified composition of patterns and their variants (1. *Vocabulary*: $V_{PatEvol}$) along with a (b) set of rules that govern the structure and semantic of relations among pattern elements (2. *Grammar*: $G_{PatEvol}$) to create a (c) sequence of

patterns (3. *Sequencing*: $S_{PatEvol}$). We formalise the structural composition of pattern language as: $PatEvol (V_{PatEvol} \times G_{PatEvol} \times S_{PatEvol}) =$

$$\left[\begin{array}{l} V_{PatEvol} = \{CLS \triangleright PAT_{\langle var_1, \dots, var_n \rangle} \} \dots (1) \\ \\ G_{PatEvol} = \{PAT_{\langle ClassifiedBy \rangle} CLS, \\ PAT_{\langle ComposedOf \rangle} OPR, \\ PAT_{\langle ConstrainedBy \rangle} CNS, \\ PAT_{\langle Evolves \rangle} ARCH, \\ PAT_{\langle hasVariant \rangle} VAR \} \dots (2) \\ \\ S_{PatEvol} = \{PAT_{1 \langle var_1, \dots, var_n \rangle} \bowtie \dots \bowtie PAT_{n \langle var_1, \dots, var_n \rangle} \} \dots (3) \end{array} \right]$$

In Section 5.1, we provide the technical details to justify graph-based formalisation of pattern language grammar. The pattern meta-model is extended to derive language grammar by adding two elements (a) possible variants of a pattern ($PAT \langle hasVariant \rangle VAR$) and (b) pattern relations expressed as ($PAT_i \langle follows \rangle PAT_j$).

5.1 Graph-based Formalisation of the Language Grammar

The language grammar ($G_{PatEvol}$) is represented as the structural composition of pattern model and semantic relationships among elements in a pattern meta-model – outlined in Listing 1. One of the most critical concerns in pattern language composition is the formalisation of the pattern language grammar [PCW05] [Zdu07]. We provide a concrete representation of the language grammar that allows us to better explain pattern specification and pattern selection aspects in this paper. We prefer graphs to formalise language grammar for:

- *Establishing Static and Dynamic Relationships*: in contrast to some predefined interconnection among patterns [PCW05] [Zdu07], attributed graph-based modeling [EPT04] [UEH⁺01] allows capturing the structure and semantics of pattern relationships. The attributed graph describes the pattern language with individual pattern as attributed nodes and pattern relationships as attributed edges (Listing 1). Graph-based modeling enables pattern relations expressed as dynamic creation or deletion of edges (relations) among nodes (patterns).
- *Pattern Matching and Selection*: we represent the individual patterns as graph nodes and exploit *sub-graph isomorphisms* [CMR⁺96] [JCZ04] (based on node matching) to select individual patterns (i.e., nodes) from language (i.e., graph).
- *Visualising Pattern Composition and Relations*: enables abstracting a complex pattern hierarchy. Pattern visualisation greatly helps with analysing pattern structure to evaluate the possible consequences and alternatives in a given evolution context.
- *Graph-based Topology of Patterns*: to define possible relationships among patterns in the language, a graph-based structure enables a flexible mechanism to search and retrieve patterns efficiently by means of graph traversal techniques.

We formalise the pattern grammar as an attributed graph (AG) with nodes and edges typed over an attributed typed graph (ATG) [EPT04]. An overview of the mapping between the elements of pattern template and the graph-based notation are provided in Listing 1 using the Graph Modeling Language. The pattern language grammar as a nested attributed graph is expressed as 6-tuple below: $G_{PatEvol} = \langle G_{TMP}, N_{CLS}, G_{PAT}, N_{CMP}, E_{SEQ}, N_{REL} \rangle$

1. *Pattern Template* [G_{TMP}] - outer graph (Line 02 - 36)
2. *Pattern Classification* [N_{CLS}] - outer graph node (Line 04 - 33)
3. *Pattern Specification* [G_{PAT}] - inner graph (Line 07 - 32)
4. *Pattern Composition* [N_{CMP}] - inner graph nodes (Line 09 -29)
5. *Pattern Composition Relationships* [E_{SEQ}] - inner graph edges (Line 15 - 31)
6. *Pattern Sequencing in the Language* [N_{REL}] - outer graph edge (Line 34 - 35)

Graph-based Template for Language Grammar Specification A language grammar also provides a structured template to capture patterns and their relations [HAZ07] [Zdu07]. We map each of the elements of the pattern meta-model (cf. Figure 1) and extend it to the possible relationships among the pattern elements in the template provided in Listing 1. To enable compositional semantics for pattern elements, we address binary compositional relationships among a change pattern (P) and its constituent element (E) given as a tuple $P \xleftarrow{relation} E$: $\langle ClassifiedBy, ComposedOf, ConstrainedBy, Evolves, hasVariant, follows \rangle$. For example, the relation between a pattern and change operators is specified as: Operators $\xleftarrow{composedOf}$ Pattern.

```

01: <graphml>
02:   <graph id="PatternTemplate" edgedefault="directed">
03:     <desc> Graph-based Representation of Change Pattern Template </desc>
04:     <node id = "CLS1">
05:       <desc> The graph node provides pattern classification </desc>
06:     </node>
07:     <graph id="ChangePattern" edgedefault="directed">
08:       <desc> A nested graph to represent individual pattern </desc>
09:       <node id = "PAT1">
10:         <desc> Pattern specific details </desc>
11:       </node>
12:       <node id = "ARCH1">
13:         <desc> Represent the architecture elements affected </desc>
14:       </node>
15:       <edge id = "Evolves" source = "PAT1" target="ARCH1">
16:         <desc> Pattern Evolves Architecture </desc>
17:       </edge>
18:       <node id = "CNS1">
19:         <desc> Specifies the enforced constraints </desc>
20:       </node>
21:       <edge id = "isConstrainedBy" source = "PAT1" target="CNS1">
22:         <desc> Pattern Constrained By Constraints </desc>
23:       </edge>
24:       <node id = "OPR1">
25:         <desc> Representing the Change Operationalisation </desc>
26:       </node>
27:       <edge id = "isComposedOf" source = "PAT1" target="OPR1">
28:         <desc> Pattern Composed of Operators </desc>
29:       </edge>
30:       <node id = "VAR1">
31:         <desc> Possible pattern variants </desc>
32:       </node>
33:       <edge id = "hasVariant" source = "PAT1" target="VAR1">
34:         <desc> Pattern has a Variant </desc>
35:       </edge>
36:     </graph>
37:   </graphml>

```

Pattern Composition						
Pattern Template Line 02	Classification Line 04 - 36	Change Pattern Line 07 - 32	Architecture Line 12 - 14	Constraints Line 17 - 19	Operators Line 22 - 24	Variants Line 27 - 29

Pattern Relationships						
PAT<Evolves>ARCH Line 15 - 16		PAT<isConstrainedBy>CNS Line 20 - 21		PAT<isComposedOf>OPR Line 25 - 26		PAT<hasVariant>VAR Line 30 - 31
PAT<Follows>PAT Line 34 - 35						

Listing 1. Graph-based Template for Change Pattern Specification (GraphML notation).

1. **[Classifies:** CLS $\xleftarrow{ClassifiedBy}$ PAT] – defines the classification of change pattern instances in the pattern language grammar. Pattern classification therefore defines a logical grouping of change patterns based on their impact of change on architecture model.
2. **[ComposedOf:** OPR $\xleftarrow{ComposedOf}$ PAT] – defines the change operational composition in a given pattern instance. It reflects the type of change operations (Add(), Rem(), Mod()) and the composition (atomic, composite operators) that specify the change operationalisation.
3. **[ConstrainedBy:** CNS $\xleftarrow{ConstrainedBy}$ PAT] – defines a set of constraints that ensure the structural integrity of architecture models before and after change pattern application. Constraints specify the preconditions (architecture model before pattern application), the invariants (preserving the compositional hierarchy of architecture model during change) and post-conditions (architecture model after pattern application).

4. [Evolves: ARCH $\xleftarrow{\text{Evolves}}$ PAT] – defines the application of a change pattern on a given architecture model. Additional details about the composition hierarchy in architecture models are presented in Section 6.

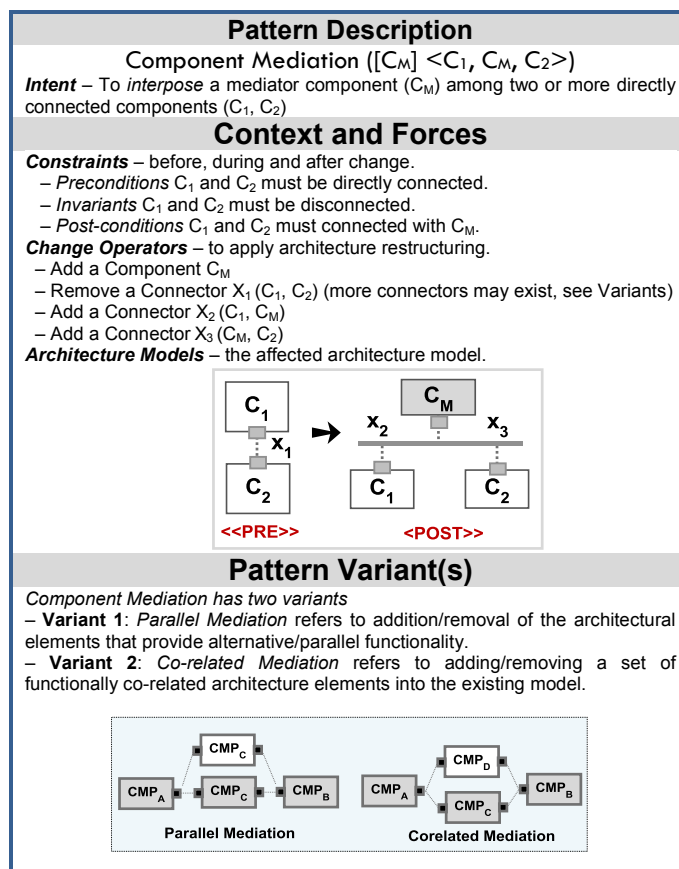
5. [hasVariant: (VAR $\xleftarrow{\text{hasVariant}}$ PAT)] – defines the relationship between a pattern and its variants. The variant of a pattern has the same structure and semantics as a pattern; it represents the variations among the possible implementations of a pattern.

6. [Follows: PAT_J $\xleftarrow{\text{follows}}$ PAT_K] – defines the sequence among two change patterns PAT_K follows PAT_J. To develop pattern relations, patterns must be applied in a specific order defined by pattern sequences. The sequence <PAT_J, PAT_K, PAT_L> means a pattern PAT_J is selected before pattern PAT_K, which itself is selected before PAT_L.

5.2 Language Vocabulary – Change Pattern and their Variants

The language vocabulary ($V_{PatEvol}$) is a classified (*CLS*) composition of change pattern (*PAT*) instances and their possible variants (*VAR*). We specify discovered pattern instances in a pattern template developed by following the language grammar (Listing 1). Based on guidelines in [HAZ07], we specify patterns by capturing a) *Pattern Description*, b) *Context and Forces*, and c) *Pattern Variants* detailed in pattern template below.

Pattern 1 - Component Mediation Pattern



Pattern 2 – Functional Slicing Pattern.

Functional Slicing($[C] \langle C_1, C_2 \rangle$)

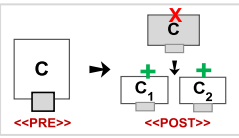
Intent – To *split* a component (C) into two or more components (C_1, C_2) for functional decomposition of C.

Constraints – before, during and after change.

- *Preconditions* C already exists in the architecture.
- *Invariants* N/A.
- *Post-conditions* C removed, C_1 and C_2 must be added.

Change Operators – to apply architecture restructuring.

- Add a Component C_1 by splitting C
- Add a Component C_2 by splitting C
- Remove a Component C



Pattern 3 – Functional Unification Pattern.

Functional Unification($\langle C_1, C_2 \rangle [C]$)

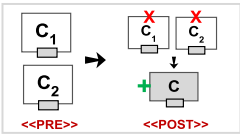
Intent – To *merge* two or more components (C_1, C_2) into a single component (C) for functional unification of (C_1, C_2).

Constraints – conditions before, during and after change.

- *Preconditions* C_1 and C_2 already exist in the architecture.
- *Invariants* N/A
- *Post-conditions* C_1 and C_2 removed, C is added.

Change Operators – to apply architecture restructuring.

- Add a Component C
- Remove a Component C_1
- Remove a Component C_2



Pattern 4 – Active Displacement Pattern.

Active Displacement($\langle C_1:C_2 \rangle, \langle C_1:C_3 \rangle [C_2:C_3]$)

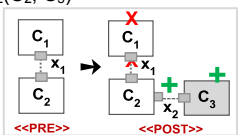
Intent – To *replace* an existing component (C_1) with a new component (C_3) while maintaining the interconnection with existing component (C_2).

Constraints – conditions before, during and after change.

- *Preconditions* C_1 and C_2 must be directly connected.
- *Invariants* C_2 exists in architecture, C_1 is removed.
- *Post-conditions* C_2 connected to a new component C_3 .

Change Operators – to apply architecture restructuring.

- Remove a Component C_1
- Remove a Connector $X_1(C_1, C_2)$
- Add a Component C_3
- Add a Connector $X_2(C_2, C_3)$



Pattern 5 – Child Creation Pattern.

Child Creation ([C] <X₁:C>)

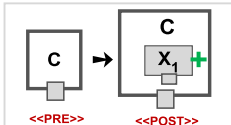
Intent – To *create* a child component (X₁) inside an atomic component (C).

Constraints – conditions before, during and after change.

- *Preconditions* component C is an atomic component.
- *Invariants* N/A.
- *Post-conditions* X₁ is a child component of C (C is Composite).

Change Operators – to apply architecture restructuring.

- Add a Component X₁
- Move in Component X₁ inside a Component C



Pattern 6 – Child Adoption Pattern.

Child Adoption (<C₁:X₁, C₂>, <C₁, C₂:X₁>)

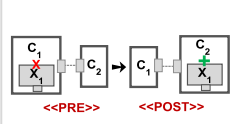
Intent – To *adopt* a child component (X₁) from a composite component (C₁) to an atomic component (C₂).

Constraints – conditions before, during and after change.

- *Preconditions* X₁ is a child inside composite C₁.
- *Invariants* X₁ is removed from C₁.
- *Post-conditions* X₁ is added in component C₂.

Change Operators – to apply architecture restructuring.

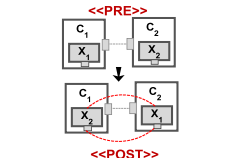
- Remove a Component X₁ from Component C₁ (C₁ is atomic)
- Add a Component X₁ into Component C₂ (C₂ is composite)



Pattern 7 – Child Swapping Pattern.

Child Swapping ([X₁:C₁], [X₂:C₂] <X₂:C₁>, <X₁:C₂>)

1. **Intent** – To *swap* the child components (X₁, X₂) from composite components (C₁, C₂).
2. **Constraints** – conditions before, during and after change.
 - *Preconditions* X₁ is a child of composite component C₁, X₂ is a child of composite component C₂.
 - *Invariants* C₁ and C₂ must be moved out of their parents C₁ and C₂.
 - *Post-conditions* X₂ is a child component of C₁, X₁ is a child component of C₂
3. **Change Operators** – to apply architecture restructuring.
 - Remove a Component X₁ from Component C₁
 - Add a Component X₁ into Component C₂
 - Remove a Component X₂ from Component C₂
 - Add a Component X₂ into Component C₁



5.3 Language Sequencing – Change Pattern Relationships

Once a grammar (Section 5.1) and vocabulary (Section 5.2) are specified, we can derive pattern sequences conforming to the language, as presented in Figure 3. More specifically, the pattern language enables pattern sequence similar to the natural language where the grammar provides the structure for generating sentences. An important question is ‘*why we choose a particular sequence of patterns among the possible alternative sequences in the language*’. In the literature, pattern sequencing is derived based on a pattern hierarchy [PCW05] (e.g., generalised patterns must be on top of specialised patterns). For example, composition, decomposition, or replacement of elements is considered as a specialisation of addition, removal, or modification of elements. Another solution for language sequencing offers annotated grammar [Zdu07] with the Question Option Criteria methodology [MYB⁺91]. In contrast to pattern hierarchy [PCW05] and annotated grammar [Zdu07], we propose *static sequences* and *dynamic sequences*.

– *Static Sequence of Patterns* allows a pre-determined, mostly manual analysis of the domain (by *pattern author* or *language creator*) to establish a fixed sequence among the patterns. An example of the static sequencing is provided in Figure 3. In Figure 3, we derive a sequence that is interpreted as **ComponentMediation** <follows> **ActiveDisplacement**: “*if the replacement of a component is required*”. A pattern sequence annotation in the language (Listing 1 Line 34-35) is specified as:

```
<edge id="Follows" source="ComponentMediation" target="ActiveDisplacement">
  <desc> if the replacement of a component is required </desc>
</edge>
```

Limitations of Static Sequencing – static sequencing is a rigid structure of pattern relations in the language with minimal flexibility. This could be particularly limiting in a context where the exact sequences of patterns depend on some arbitrary evolution scenario. For example, in many situations, we might need the application of **ActiveDisplacement** pattern before applying **ComponentMediation**. This and many other similar situations ignore static sequences, as a static sequence represents a specific organisation of patterns that must be dynamically adjusted.

– *Dynamic Sequence of Patterns* – in contrast to static ones, they provide a flexible sequence by means of dynamic relationships among patterns. To support dynamic sequences, we follow the design-space analysis for patterns [Zdu07] based on the Question Option Criteria (QOC) [MYB⁺91]. Details and a practical case of pattern sequencing are presented in Section 7. For example, the intent of a pattern language user is to enable component composition but he/she is unaware which pattern to select. By following QOC, patterns can be applied in sequence guided by the pattern language user (selecting one pattern at a time).

1. *Question* – How to compose an *atomic* component into a *composite* one?
2. *Option* – **ChildCreation** pattern enables component composition.
3. *Criteria* – The consequence of **ChildCreation** is the composition of an atomic component into a composite component (composed of one or more child/sub component).

We propose that the **vocabulary** of pattern language continuously evolves as new patterns are discovered and integrated in the language by following language grammar (cf. Section 5.1). Once a new pattern is discovered [AJP12b] [AJA⁺12], it is up to the user or more specifically pattern author to provide a *name* and *intent* for the new pattern (confirming to language **grammar** in Listing 1). As the last step, establishing the relation(s) of newly discovered pattern with existing patterns is achieved with pattern **sequences** (already discussed above).

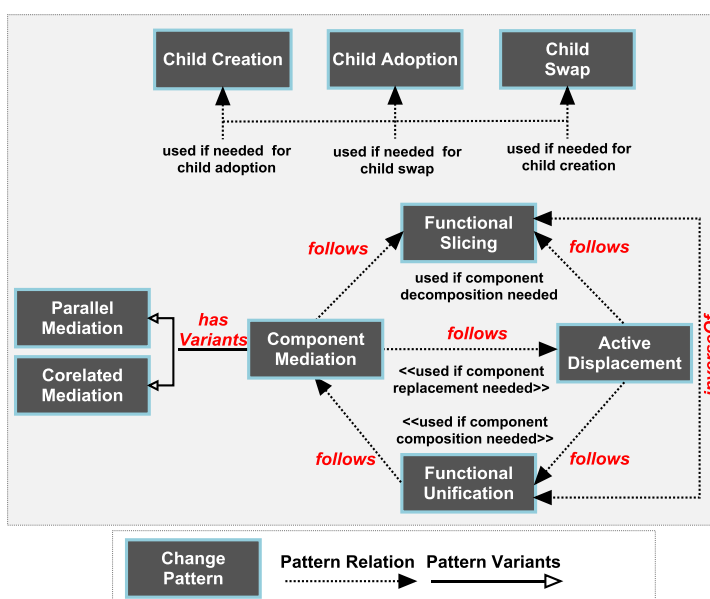


Figure 3. An Overview of Change Pattern Language.

6 Application Domain of the Pattern Language

The applicability (a.k.a. *application domain*) of the proposed pattern language is limited to component-based software architecture (CBSA) models and their evolution. In this section, we focus on presenting an overview of CBSA for an EBPP (Electronic Billing Presentment and Payment) case study³ and discuss the EBPP evolution scenarios. The case study and evolution scenarios introduced here are used to clarify pattern selection and pattern application aspects in Section 7.

6.1 Description of Composition-Based Architecture Models

Architectural descriptions of a component-based architecture model as an attributed graph are presented in Figure 4. We present the architecture meta-model as an attributed typed graph (ATG) [EPT04], while a possible architectural instance (Figure 4) is represented as an attributed graph (AG) that is typed over an ATG. We prefer graph-based modeling mainly because: *we model architecture as a graph and exploit graph transformations to evolve the architecture in a formal and automated way* [CMR⁺96]. More specifically, exploiting the

³ <http://www.nacha.org/ebpp>

Double Push Out graph transformation [CMR⁺96] maintains the structural integrity of evolving architecture (further details in Section 7).

In Figure 4, the CBSA model represents *Architectural Configuration* (CFG: `cfg_Payment`) as a composition of computational *Components* and their *Connectors*. Furthermore, each component must contain a *Port* for communication and **atomic components** could be composed into **composite** ones. Connectors must contain *Endpoints* for component binding.

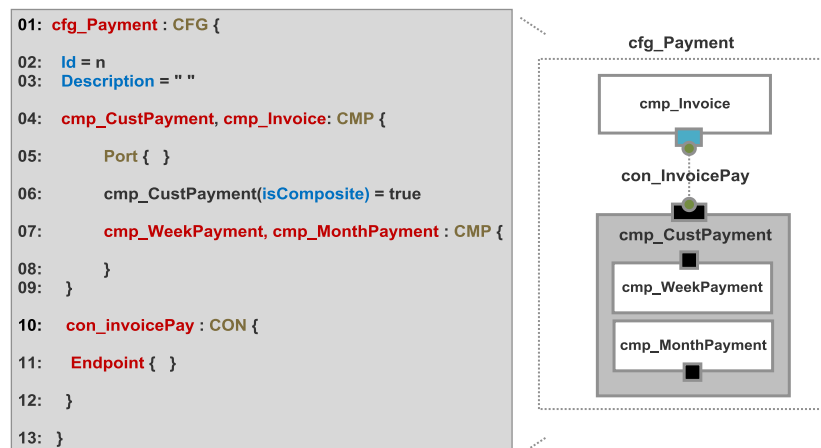


Figure 4. Structured Description of CBSA Model.

6.2 Component and Connector Architectural View for EBPP

A high-level component and connector view of the EBPP is presented in Figure 5. For illustrative reasons, we abstract the details about data store (DS) and user interface (UI) layers and focus on architectural layers modeling components and connectors [MT00] [KCM07].

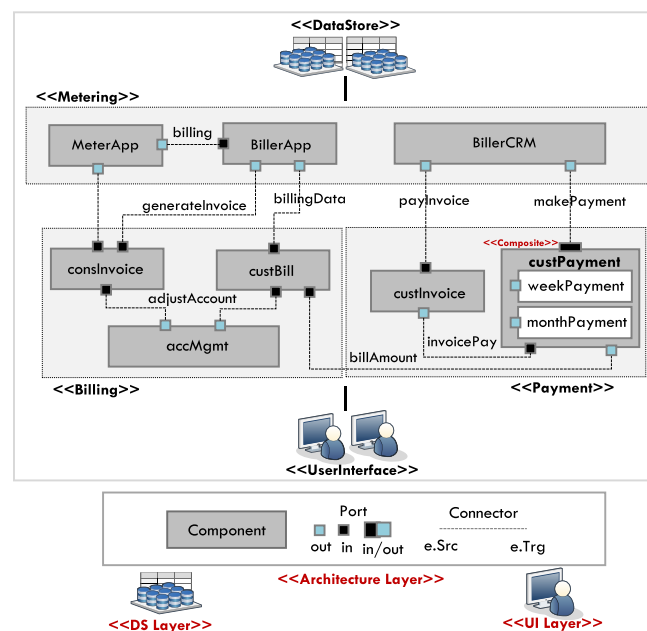


Figure 5. Architectural View for EBPP (before Evolution).

Architectural configurations represent *Metering* (to provide meter information for customer's consumption), *Billing* (to handle customer billing), and *Payment* (to manage customer payments corresponding to the billing amount). We are interested in component, connectors, and the interaction (messaging) that exists among the components.

- **Components (CMP)** represents the first-class entities as computational elements or data stores of the EBPP architecture model in Figure 5. Component type classification is:

1. *Atomic Component* - is the fundamental type of a component that could not be decomposed. Atomic components in *Metering* configuration are *BillerCRM*, *BillerApp* and *MeterApp*.

2. *Composite Component* - represents a component that contains an internal architecture as a sub-configuration of components and connectors inside composite component. The only example of composite component in is *custPayment* that has *weekPayment* and *monthPayment* as its children or sub-component.

- **Connectors (CON)** are responsible for message passing among the component ports. Unlike composite components, EBPP architecture has only *atomic connectors* for component interconnection. Example of a connector-based message passing among *BillerCRM* (port:out - source) and *custPayment* (port:in - sink) components is expressed as *makePayment* connector.

6.3 Evolution Scenarios for EBPP Architecture

We look at some evolution scenarios to demonstrate desired changes in existing architecture model for the EBPP system. We adopt the Architecture Level Modifiability Analysis (ALMA) [BLB⁺04] method for *identification* and *analysis* of EBPP architecture evolution as presented in Figure 6. The motive for using ALMA method is to systematically:

- *Select* architecture evolution scenario (evolution pre-condition),
- *Evaluate* impacts of selected scenarios on existing architecture,
- *Interpret* results of scenario execution (evolution post-condition).

In addition, the results of pattern-base changes on architecture model can be evaluated and interpreted using ALMA.

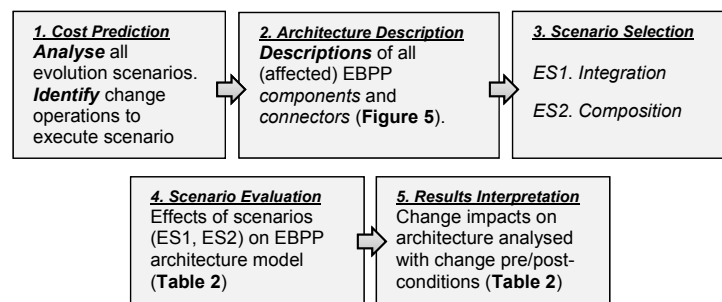
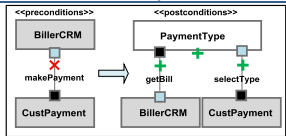
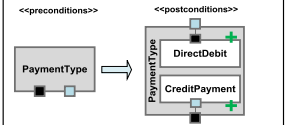


Figure 6. Overview of ALMA for Selection, Evaluation and Interpretation of Scenarios.

Once we have (a) analysed evolution cost (Figure 6) and (b) specified the architecture (Figure 4), we now present (c) selected scenarios along with (d) their evaluation and (e) change impact interpretation in Table 2. A set of evolution scenarios are presented in Table 2 following ALMA [BLB⁺04]. Key characteristics and evolution-centric aspects of CBSA are:

- *Composite Change Execution* must abstract atomic changes (*addition, removal, and modification of component and connectors*) into composite ones that allow *integration, composition, and replacement* of a set of architecture elements [GTO⁺08] [AJA⁺12].
- *Evolution Reuse* is a key characteristic that must enable a generic, reuse-driven change for recurring evolution problems in component-based architecture models [BGS12].
- *Consistency of Architecture Models* ensures structural integrity and composition constraints of architecture are preserved before (*pre-conditions*) and after evolution (*post-conditions*).

Table 2. Selection, Evaluation and Interpretation of Scenarios with guidelines in ALMA.

[X = Removal] [+ = Addition] [<preconditions> Evolution <postconditions>]		
Scenario Selection	Scenario Evaluation	Results Interpretation
ES1 - [...] to integrate a component PaymentType that facilitates the selection of a payment type mechanism between the directly connected components BillerCRM and CustPayment.	EBPP architecture is modified with addition of a new components and two connectors to mediate customer billing and payments: opr:=ADD(PaymentType ∈ CMP) opr1:=ADD(<getBill, selectType> ∈ CON)	
ES2 - [...] to compose the PaymentType component with DirectDebit and CreditPayment child components that allows a customer to avail-of flexible options for billing payments.	The internal architecture of PaymentType is modified with addition of two child components DirectDebit and CreditPayment opr:=ADD(DirectDebit ∈ CMP) opr:=ADD(CreditPayment ∈ CON)	

7 Pattern Language for Evolution in CBSA

After presenting the pattern language composition (Section 5) and architecture evolution scenarios (Section 6), we now focus on the pattern language application to support architectural evolution of EBPP. Language support for evolution refers to exploiting a collection of patterns as generic, repeatable solutions to recurring architecture evolution problems. The patterns from the language can be selected and applied in a sequential fashion to support an incremental evolution [HZ06] in CBSAs.

7.1 Change Patterns as Evolution-centric Reuse Knowledge

In the proposed pattern language, architecture evolution-reuse knowledge is expressed as pattern collection that enables mapping among the problem-solution view to enable reusable evolution strategies (cf. Section 4, 5). An overview of the pattern-based evolution is presented in Figure 7.

To enable pattern-driven reusable change execution, we adopt the design space analysis [MYB⁺91] [MM95] for a systematic pattern selection from language collections. Design space analysis is a methodology to address design-related problems in human-computer-interaction (HCI) [MM95]. Here, we utilise *design space analysis* for an (a) explicit representation of alternative evolution strategies (available patterns) and (b) the rationale for choosing among available strategies (selected patterns). In Figure 7 we illustrate that:

– *Problem Space* represents evolution scenarios that we identified from the EBPP case study in Table 2.

A Pattern Language for the Evolution of CBSAs

- *Problem-Solution Map* represents the patterns and their relations in the language to provide a mapping of evolution scenarios and their solution as pattern instances (Section 5).
- *Solution Space* represents pattern-driven reuse to guide architecture evolution that is the focus of this section.

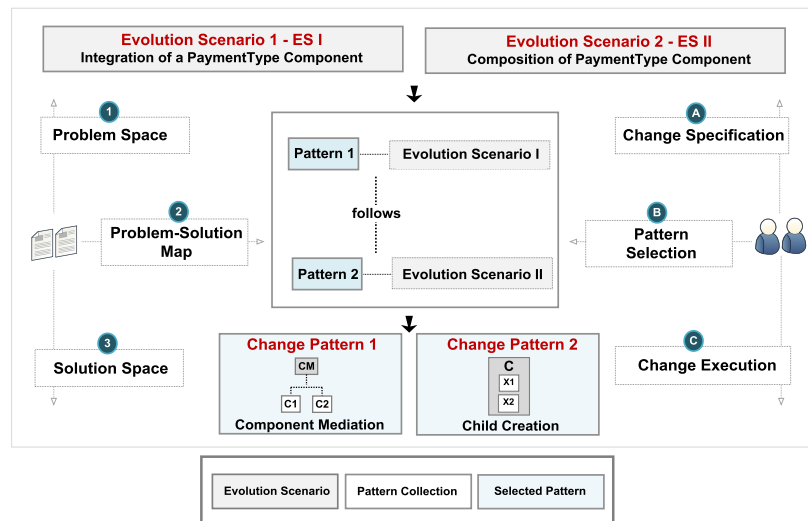


Figure 7. Overview of the Mapping among Problems (*Scenarios*) to their Solutions (*Patterns*).

7.2 Pattern Selection with Design Space Analysis

In a technical context, problem-solution mapping represents the pattern selection problem based on a given evolution context. First, we look at the Question-Option-Criteria [MYB⁺91] (using design space analysis [MM95]) that allow us to resolve the pattern selection problem to enable pattern-driven evolution. We illustrate pattern selection in Figure 8 by illustrating the selection of Component Mediation patterns (cf. Section 5 – Pattern 1). Figure 8 represents a visualisation of the 3-step QOC-based pattern selection process:

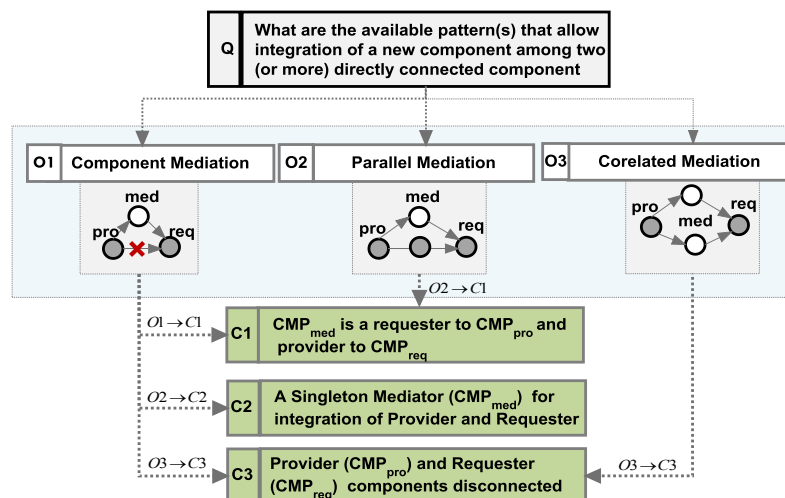


Figure 8. QOC Methodology for Pattern Selection.

1. *Question* – allows representation of problem space that allows a declarative specification for intent of change, e.g; *What are the available pattern(s) to integrate a component **PaymentType** that facilitates the selection of a payment type mechanism between the directly connected components **BillerCRM** and **CustPayment**?*

2. *Options* – enables problem-solution mapping with selection of the most appropriate pattern from language collection, e.g; *The available pattern for integration is **Component Mediation** that has two variants **Parallel Mediation** and **Corelated Mediation** patterns.*

3. *Criteria* – defines analysing the solution space to allow evolution of architecture by satisfying the given criteria, e.g; *The application of **Component Mediation** allows a mediator component integrated among two directly connected components.*

7.3 Pattern Application with Graph-Transformation

After an overview of pattern selection, we now focus on pattern application for architecture evolution that is guided by graph-transformation [CMR⁺96]. In the context of model-driven evolution of software architectures [Gra07], our solution models architecture as a graph (cf. Section 6) and exploits graph transformation that is guided by change patterns to evolve architectures. This means specification of architecture models as graph allows us to exploit graph transformation for architectural evolution [CMR⁺96].

To support architectural evolution with pattern language, we primarily focus on (a) *enabling change reuse* and (b) *maintaining the structural consistency of architecture* before and after change execution. More specifically, during change execution change operationalisation is abstracted as declarative graph transformation rules. Evolution in the context of composition-based architecture abstracts atomic changes into a set of composite change operations. The atomic change operations (*Add()*, *Remove()*, *Modify()*) on architectural elements (*components* and *connectors*) must be abstracted into reusable, composite and domain specific changes. Composite-domain specific changes include *Integrate()*, *Replace()*, *Decompose()*, *Split()*, *Merge()* etc. of architectural components and connectors in CBSAs.

Evolution Scenarios - In the existing functional scope of the case study (Section 6.3, Table 2), 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 the following, we illustrate the role of **ComponentMediation** followed by **ChildCreation** pattern to allow (a) the *integration of a component **PaymentType*** (ES1) and (b) the creation of its child components **DirectDebit** or **CreditPayment** (ES2).

Evolution Scenario I - *to integrate a component **PaymentType** that facilitates the selection of a payment type (direct debit, credit payment) mechanism among the directly connected components **BillerCRM** and **CustPayment**.*

In the following, we illustrate pattern-based evolution that follows a three-step process: *Change Specification*, *Pattern Selection* and *Change Execution* presented in Figure 7.

Step I – Change Specification - Questions

We specify the change rule along with architectural pre-post-conditions using GML [UEH⁺01]. A declarative specification allows architect to represent a syntactical context of architectural change that contains the (a) source architecture model

(Source<ArchitectureModel>: as *Preconditions*), (b) typed architecture elements (*ArchitectureElement* \in *ElementType*) that must be added, removed, or modified, and (c) anticipated target architecture (Target<ArchitectureModel>: as *Postconditions*). Change rule is formally expressed as:

```
<node id = "Change Rule">
  <desc> Specification of Change Rule </desc>
  <data key="ChangeRule"> Integration </data>
  <data key="Operation"> ADD </data>
  <data key="ArchitectureElement"> PaymentType </data>
  <data key="ElementType"> Component </data>
</node>
```

Step II – Pattern Selection – Options

To select an appropriate pattern, we query the pattern language based on pattern-specific conditions. These conditions are expressed as preconditions and post-conditions that must be satisfied to preserve the structural integrity of the overall architecture and individual elements during change execution.

– The **precondition(s)** represent the context of architectural elements before change execution. In Figure 9a, the precondition (*pre*) specifies the exact sub-architecture *makePayment*(BillerCRM, CustPayment) that must be changed in the source architecture (*source*). To apply changes, we must find an exact structural match m_s of preconditions in the architecture such that $m_s: pre \rightarrow source$ as in Figure 9a. Figure 9 follows the Double-Push-Out (DPO) [CMR⁺96] approach for graph transformation. The DPO graph transformation allows the (a) *source architecture* (graph) to be transformed into the (b) *target architecture* (graph) by using an *intermediate architecture* (graph). We represent the source, intermediate and target architecture as preconditions, *invariants* and *postconditions* of transformation. For example, preconditions are expressed as (cf. Figure 9a):

```
<node id = "Preconditions">
  <desc> Specification of Preconditions </desc>
  <data key="ArchitectureElement"> BillerCRM </data>
  <data key="ElementType"> CMP </data>
  <data key="ArchitectureElement"> CustPayment </data>
  <data key="ElementType"> CMP </data>
  ....
</node>
```

– The **invariants** represent the architectural structure that is never changed during evolution. This is represented as Figure 9b, the intermediate architecture $m_i: inv \rightarrow intermediate$ with Double-Push-Out (DPO) graph transformations [CMR⁺96].

```
<node id = "Invariants">
  <desc> Target Architecture Model</desc>
  <data key="ArchitectureElement"> BillerCRM </data>
  <data key="ElementType"> CMP </data>
  <data key="ArchitectureElement"> CustPayment </data>
  <data key="ElementType"> CMP </data>
  ....
</node>
```

– The **postcondition(s)** specify the context of evolved architectural elements as a result of the change execution. After applying changes on specified elements the overall architectural

structure must be preserved. To include the modified architecture elements in the target architecture (*target*) an exact structural match m_t of postconditions in target architecture must exist $m_t: post \rightarrow target$ (Figure 9c) expressed as:

```

<node id = "PostConditions">
  <desc> Target Architecture Model</desc>
  <data key="ArchitectureElement"> BillerCRM </data>
  <data key="ElementType"> CMP </data>
  <data key="ArchitectureElement"> PaymentType </data>
  <data key="ElementType"> CMP </data>
  <data key="ArchitectureElement"> CustPayment </data>
  <data key="ElementType"> CMP </data>
  ....
</node>

```

Step III – Change Execution - Criteria

Once an exact instance of *preconditions* in a *source architecture* is identified, the pattern language is queried with pre-conditions and post-conditions that enable the retrieval of the appropriate pattern that provides the potential reuse of change operationalisation to enable architectural evolution (cf. Figure 8). The query matches the specified change pre-conditions and post-conditions to retrieve the pattern definition. Figure 9 illustrates the retrieved instance of ComponentMediation pattern. In addition, **pattern instantiation** involves the labeling of generic elements in specifications with labels of concrete architecture elements. For example, in Figure 9a, the connector instance *makePayment* that is missing in the change post-conditions is removed from the *source architecture*. The newly added instance(s) of component *PaymentType* and connector *getType*, *makePayment* are the candidates for addition into *source* to obtain *target* in Figure 9c and is represented as:

```

<node id = "ChangeOperations">
  <desc> Change Operationalisation</desc>
  <data key="ChangeOperator"> Add </data>
  <data key="ArchitectureElement"> PaymentType </data>
  <data key="ElementType"> CMP </data>
  <data key="ChangeOperator"> Remove </data>
  <data key="ArchitectureElement"> makePayment </data>
  <data key="ElementType"> Connector </data>
  ....
  ....
</node>

```

– **Change Operationalisation** We provide a brief overview of the change execution that is facilitated using the DPO construction [CMR⁺96]. In Figure 9, the order of change operations is insignificant and the sequence is presented as it appeared in the given pattern instance.

– **Deletion:** In Figure 9b, *Source/Intermediate* describes the architecture elements to be deleted from the *source architecture*.

For example, the connector *makePayment* is removed from the *BillerCRM* and *CustPayment*. The intermediate architecture *is* obtained from the source architecture for elements which are a pre-image in *Source*, but not in *Intermediate*

– **Addition:** In Figure 9c, *Target/Intermediate* described the part that must be added in *Source* to obtain *Target* during change execution. In Figure 9c, the component *PaymentType* is added with connector *selectType* and *custPay* in the architecture.

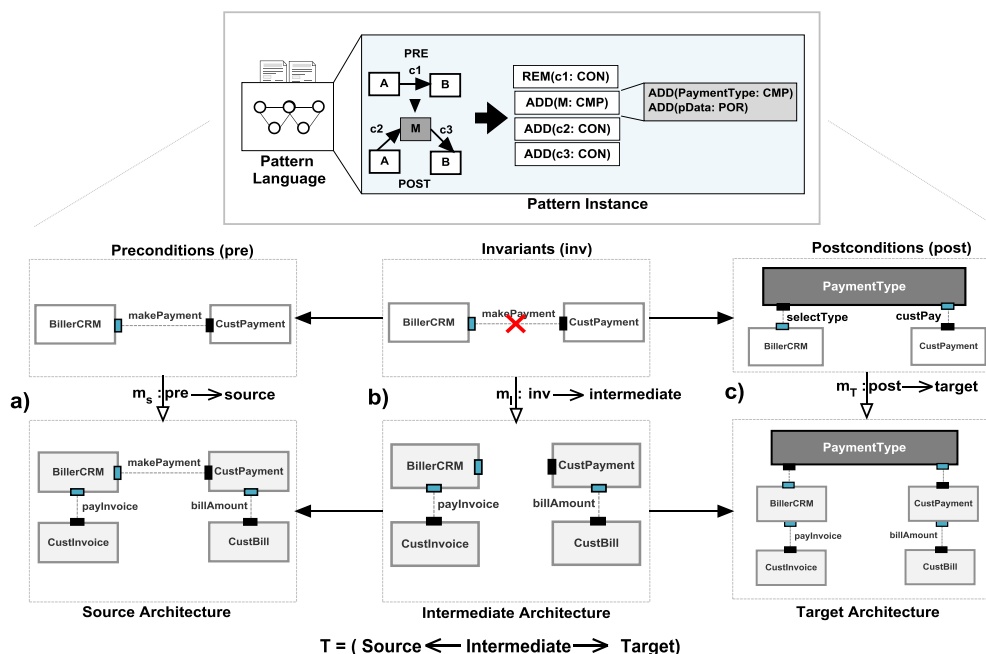


Figure 9: Pattern-based Architecture Evolution Using Graph-Transformation (DPO Approach)

Evolution Scenario II - to compose the *PaymentType* component with *DirectDebit* and *CreditPayment* child components that allows a customer to avail-of flexible options for billing payments.

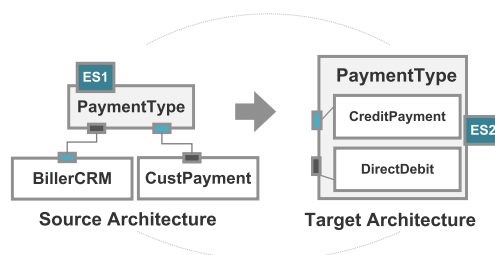


Figure 10. Child Creation Pattern to Enable Payment Type Options.

After applying the *ComponentMediation* pattern above, in Figure 10 again we follow the process (**Step I** Change Specification, **Step II** Pattern Selection) to select the *ChildCreation* pattern from language. The *ChildCreation* creation patterns creates the *DirectDebit* and *CreditPayment* child components into the newly added *PaymentType* component (**Step III** Change Execution) expressed as: [*ComponentMediation* follows *ChildCreation*].

8 Implementation and Discussion

In this section, first we discuss the role of the *prototype* PatEvol that facilitates semi-automation and parameterised customisation of the evolution process. We discuss preliminary evaluations of pattern discovery and pattern selection problems and future validations.

8.1 PatEvol Prototype

The prototype is presented in Figure 11 with screenshots of its interfaces explained below:

1. *Change Specification Interface* – allows an architect to declaratively specify the intent of change as the evolution rule (cf. Section 7.3). An evolution rule explicitly specifies intent of change, the architecture models to be evolved, i.e., source architecture and preconditions of the architecture model. In the functional context of the prototype, rules are specified in GML.

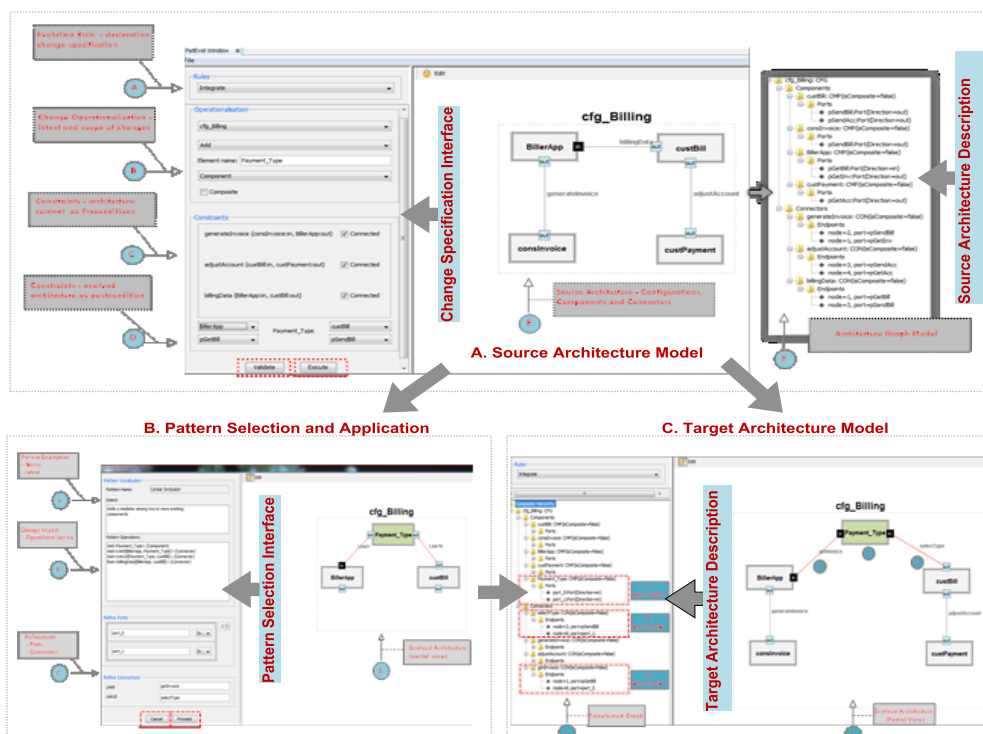


Figure 11. Prototype-Tool Support for Pattern-Driven Architecture Evolution.

2. *Architectural Descriptions* – are provided with a graph-based notation (GML) [UEH⁺01]. Architectural descriptions before and after evolution are verified with pre-/postconditions to ensure structural integrity of the architecture is preserved.

3. *Pattern Selection and Application* – patterns are expressed in the language as a nested graph. Pattern selection is enabled with design space analysis based on the QOC methodology [MYB⁺91] (cf. Section 7.2). The developed prototype (PatEvol) assists the user to specify and execute pattern-based changes with appropriate parameterisation and customisation of the process as presented in Figure 11. In Figure 11, we illustrate integration of a component among directly connected components as the evolution scenario (ES1, cf. Table 2 – also the running example in Section 7). More specifically, the prototype interface allows the user to specify the (a) *intent of change*, (b) *architecture elements in the source model* (to be added, removed or modified), and (c) *preconditions on source architecture model*.

8.2 Preliminary Evaluations

The overall solution requires evaluation of (a) *pattern discovery* (Change Mining process) as well as (b) *Pattern selection* and *pattern application* (Change Execution process). The technical details about pattern discovery are beyond the scope of this paper.

The objective is to *evaluate the solution's capability to support reuse-driven change execution by means of change patterns*. We claim that, 'if architecture changes could be *specified declaratively*, pattern collection in the language *abstract complex operational details* to enable *generic, reusable, off-the-shelf* change execution'. Evaluation about algorithmic complexity and pattern validation process are detailed elsewhere [AJP12b] [AJA⁺12].

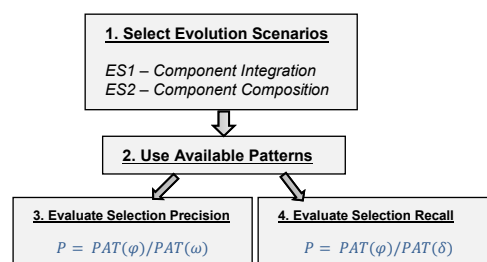
– *Experimental Setup to Evaluate Pattern Selection*: We summarise the details of the experimental design regarding the precision and recall of the pattern selection below. Evolution scenarios are selected from Table 2 identified using ALMA [BLB⁺04]. The universal search space for pattern selection is represented as pattern language comprising of a total of 9 change patterns (7 patterns and 2 variants of a pattern) in Figure 3.

Evaluation Objective: to assess the adequacy of solution in supporting pattern selection in given evolution context. We represent pattern *selection precision* and *selection recall* criteria.

– *Input(s) of Evaluation Process* are (i) evolution scenarios (cf. Table 2), and (ii) architecture change specification by the user (cf. Figure7).

– *Output(s) of Evaluation Process* is a measure of pattern selection precision and recall as:

1. *Pattern Selection Precision* (P) is defined as number of **relevant pattern instances** $PAT(\varphi)$ retrieved by a search divided by the **total number of pattern instances** $PAT(\omega)$ retrieved.
2. *Pattern Selection Recall* (R) is defined as number of **relevant pattern instances** $PAT(\varphi)$ retrieved by a search divided by the **total number of existing relevant pattern instances** $PAT(\delta)$.



Summary of Results: Based on the criteria above, a summary graph of precision and recall is presented in Figure 12. Due to a smaller search space (7 patterns and 2 variants), the recall is measured to be 0.99 approx. for all pattern instances. A high recall suggests the solution is adequate in selecting the most relevant instances from available collection. However, we experience a different behavior for precision, because identification of the exact pattern in the context of related patterns is more challenging. The corresponding values for selection precision fluctuate between 0.33 and 0.99. Whenever we query for “*component integration pattern*”, we are returned with at least three pattern instances (*Component Mediation, Parallel Mediation, and Co-related Mediation*).

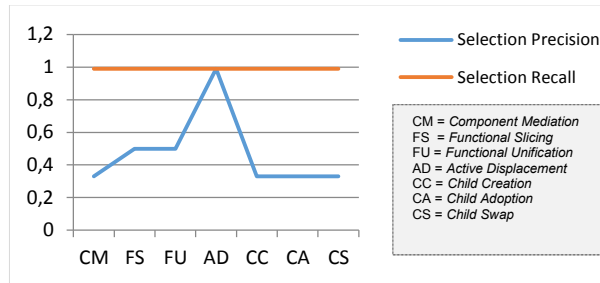


Figure 12. Precision and Recall for Pattern Selection.

8.3 Effects of Pattern Classification on Selection Precision

The classification of change patterns in Figure 13 helps us to increase the precision of pattern selection. Pattern classification enables a logical grouping of related patterns based on the types of architectural changes that a group of patterns support. The prototype in Figure 13 also allows a user to specify and classify the change. In the following, we discuss the individual elements of the prototype to specify change patterns.

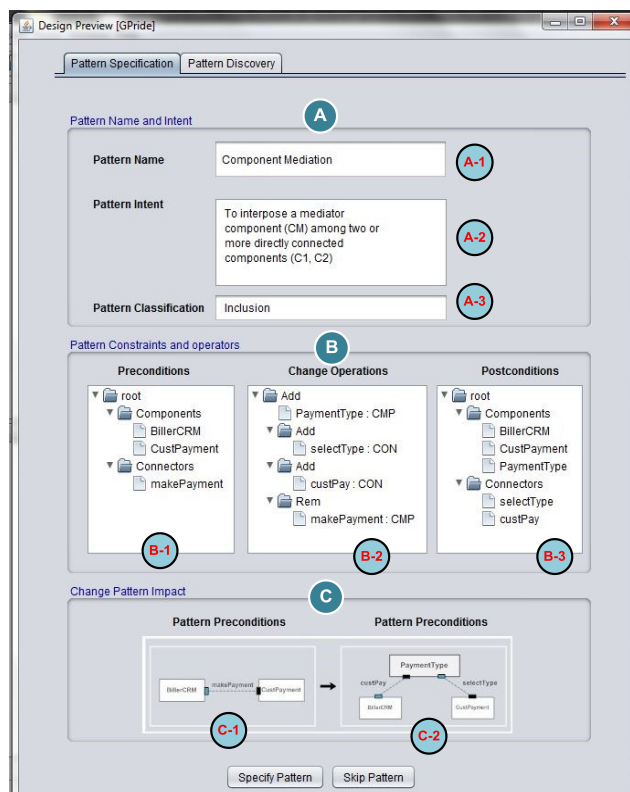


Figure 13. Screen-shot of Prototype for Change Pattern Specification and Classification.

A. Specifying Pattern Name and Intents As presented in Figure 13, the prototype allows a user to specify the *name*, *intent* and *classification* type for a pattern. Pattern name and intent are specified by a user based on the impact of change pattern. For example, in Figure 13 the visualisation of pattern preconditions and pattern post-conditions helps a user to identify that

the pattern provides **Component Mediation** among two directly connected components. The user specifies classification type of change pattern.

B. Pattern Constraints and Operations

The constraints and operations are extracted from each identified pattern and presented to the user that helps to decide about the name and intent of the pattern. The constraints are presented as preconditions and post-conditions. The change operations represent individual changes on architecture model as presented in Figure 13.

C. Change Pattern Impact

Finally, the impact of each discovered pattern is visualised to help the user to analyse the impact of change patterns before and after the application of change pattern in Figure 13.

We have classified the existing patterns into three distinct types as Composition, Association and Decomposition patterns with classification ids 1, 2, and 3 respectively, presented in Figure 14. For example, in Figure 14 the **Child Creation** pattern enables the composition of an atomic component into a composite that contains one or more child components. In the pattern language context, the classification has no effect on pattern relations – pattern(s) in one classification may be related to pattern(s) in a different classification.

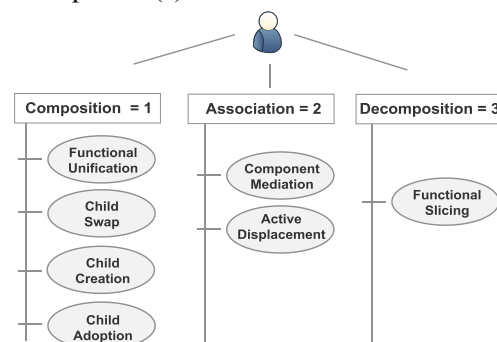


Figure 14. An Overview of Change Pattern Classification.

Pattern classification reduces the pattern search space and ultimately increasing selection precision. For example, if a user wants to select a pattern for decomposition of a composite component into atomic components; instead of searching in a space of 7 different patterns he/she can locate the Functional Slicing pattern under classification type decomposition.

A possible limitation for pattern type classification is that the user must specify the classification type in which they aim to search the patterns. If an appropriate pattern is not found, the user must rely on the primitive architectural changes. Also, if the type is not specified correctly, the appropriate pattern(s) must be searched in all the classifications which results in a lower precision – as discussed in Figure 12. Based on the pattern classification in Figure 12, a summary graph of precision and recall is presented in Figure 15. Pattern classification has no impact on selection recall factor that remains at 0.99 also highlighted in Figure 12. However, the selection precision factor is increased because the pattern search space is minimised with the pattern type classification. The selection precision for association and decomposition type patterns is 0.99. The precision for composition type pattern is 0.5 – a low precision is a consequence of the overlap of change support by Child Swap, Child Creation and Child Adoption patterns is 0.33 that is subject to further evaluations.

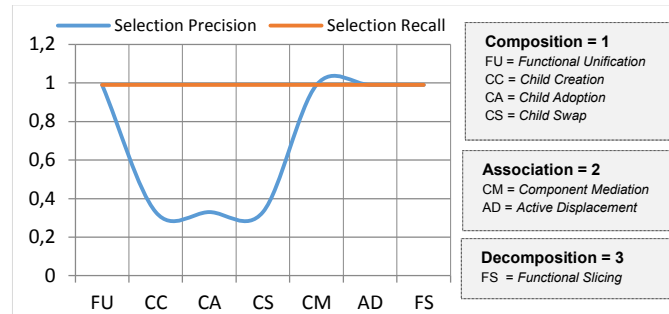


Figure 15. Precision and Recall After Pattern Classification.

Plan for Future Validations: The accuracy of pattern selection only reflects on the solution’s capability to assist and relieve an architect to retrieve the most appropriate patterns in a given evolution context. However, in future we are more concerned about a more rigorous validation by involving the software architect/designers to utilise the prototype to execute evolution scenario by accommodating more case studies. We also need to validate prototype-usage followed by expert opinion with a series of questioner for an objective evaluation of solution.

9 Conclusions and Future Research

The proposed solution is the first attempt towards promoting the pattern language as a collection of empirically discovered and connected architecture change patterns. We provide a significant extension of our previous research (on pattern discovery) to formalise and interconnect existing patterns to achieve reuse of frequent changes during evolution. The solution promotes architecture evolution as a two-step process: to leverage *architectural change mining* – discovering pattern instances from change logs – to support potential reuse during *architecture change execution*. We demonstrated that *if an architectural evolution problem can be specified declaratively, pattern-driven evolution could relieve an architect from the underlying operational concerns for executing frequent evolution tasks facilitated with change patterns*. We discussed the role of a pattern language as an explicit knowledge-base to support reuse-driven evolution in CBSAs. The ultimate contribution is:

- Enabling ‘post-mortem’ analysis of architecture evolution histories to discover operationalisation and patterns that can reused to guide architecture change management.
- Language as a formalised system-of-pattern allows problem-solution mapping to reuse generic, reusable operationalisation. The role of the pattern language is central in promoting patterns to achieve reuse and consistency in evolution for CBSA.

Currently, we have a limited number of (7+2) patterns in the language. However, language as an evolving collection requires a continuous discovery and specification of new patterns.

Future Work – In future, we will primarily focus on addressing the granularity of change execution beyond generic specifications of identified patterns. We aim to focus on classification of change patterns as *commutative* and *dependent* patterns. Such a classification can help us to analyse the extent to which architecture evolution could be parallelised (identifying dependent and independent change patterns). As part of future research, we will also focus on a systematic identification and resolution of change anti-patterns. We plan to

evaluate the solution and its prototype in a more realistic environment by involving designers or software architects. Additional case study of updating a peer-to-peer towards client-server architecture shall allow a more rigorous evaluation of architectural evolution.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

References

- [AJA⁺12] A. Ahmad, P. Jamshidi, M. Arshad, C. Pahl. Graph-based Implicit Knowledge Discovery from Architecture Change Logs. *In 7th Workshop on SHaring and Reusing Architectural Knowledge*, 2012.
- [AJP12a] A. Ahmad, P. Jamshidi, Claus Pahl. Classification and Comparison of Architecture Evolution Reuse Knowledge – A Systematic Review. *In Journal of Software Evolution and Process*, 2014.
- [AJP12b] A. Ahmad, P. Jamshidi, C. Pahl. Graph-based Pattern Identification from Architecture Change Logs. *In 10th International Workshop on System/Software Architectures*, 2012.
- [AJP12c] A. Ahmad, P. Jamshidi, and C. Pahl. Pattern-driven Reuse in Architecture-centric Evolution for Service Software. *In 7th International Conference on Software Paradigm Trends*, 2012.
- [AJP12d] A. Ahmad, P. Jamshidi, C. Pahl. Graph-based Discovery of Architecture Change Patterns from Logs. *Technical Report, School of Computing, Dublin City University*, Accessed February 2014, [online] <http://www.computing.dcu.ie/~pjamshidi/PDF/PatternDiscovery.pdf>
- [Ale99] C. Alexander. The Origins of Pattern Theory, the Future of the Theory, and the Generation of a Living World. *In IEEE Software*, 1999.
- [BGA06] S. Bouktif, Y. Guéhéneuc, G. Antonioli. Extracting Change-patterns from CVS Repositories. *In IEEE Working Conference on Reverse Engineering*, 2006.
- [BGS12] J. M. Barnes, D. Garlan and B. Schmerl. Evolution Styles: Foundations and Models for Software Architecture Evolution. *In Journal of Software and Systems Modeling*, 2012.
- [BHS07] F. Buschmann, K. Henney, and D. C. Schmidt. Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. *Wiley and Sons*, 2007.
- [BLB⁺04] P. Bengtsson, N. Lassing, J. Bosch, and H. V. Vliet, Architecture-Level Modifiability Analysis. *In Journal of Systems and Software*, vol. 69, 2004.
- [CHW07] I. Côté, M. Heisel, I. Wentzlaff. Pattern-Based Evolution of Software Architectures. *In European Conference on Software Architecture*, 2007.
- [CMR⁺96] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Lwe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. *In Handbook of Graph Grammars and Computing Graph Transformation, Volume 1: Foundations*, 1996.
- [EPT04] H. Ehrig, U. Prange, G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. *In Graph Transformations*, 2004.
- [GH04] H. Gomaa, M. Hussein. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. *In 4th Working IEEE/IFIP Conference on Software Architecture*, 2004.
- [Gra07] B. Graaf. Model-driven Evolution of Software Architectures. *In 11th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2007.
- [GTO⁺08] O. L. Goer, D. Tamzalit, M. Oussalah, A. D. Seriai. Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures. *In IEEE International*

- Computer Software and Applications Conference*, 2008.
- [GZ02] M. Goedicke and U. Zdun. Piecemeal Legacy Migrating with an Architectural Pattern Language. In *Journal of Software Maintenance: Research and Practice*, 2002.
- [HAZ07] N. B. Harrison, P. Avgeriou and U. Zdun. Using Patterns to Capture Architectural Decisions. In *IEEE Software*, 24(4): 38-45, 2007.
- [HTW⁺99] K.M. Hee, R.A. Toorn, J. Woude, P. Verkoulen. A Framework for Component Based Software Architectures. In *Software Architectures for Business Process Management*, 1999.
- [HZ06] C. Hentrich and U. Zdun. Patterns for Process-Oriented Integration in Service-Oriented Architectures. In *11th European Conference on Pattern Languages of Programs*, 2006.
- [JCZ04] C. Jiang and F. Coenen and M. Zito. A Survey of Frequent Subgraph Mining Algorithms. *The Knowledge Engineering Review*, 2004.
- [JGA⁺12] P. Jamshidi, M. Ghafari, A. Ahmad and C. Pahl. A Framework for Classifying and Comparing Architecture Centric Software Evolution. In *17th European Conference on Software Maintenance and Reengineering*, 2012.
- [KCM07] H. Kagdi, M. L. Collard, J. I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. In *the Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 19, No. 2, pp. 77-131, 2007.
- [KZ07] H. Kampffmeyer and S. Zschaler. Finding the Pattern you Need: The Design Pattern Intent Ontology. In *10th International Conference on Model Driven Engineering Languages and Systems*, 2007.
- [Leh96] M. Lehman, Laws of Software Evolution Revisited. In *Software Process Technology, LNCS* 1996.
- [MM95] A. MacLean and D. McKerlie. Design space analysis and use representations. In Scenario-based Design: Envisioning Work and Technology in System Development. *John Wiley & Sons*, 1995.
- [MRT99] N. Medvidovic, D. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-based Software Development and Evolution. In *21st International Conference on Software Engineering*, 1999.
- [MS08] T. Mens and S. Demeyer. *Software Evolution*. Springer, 2008.
- [MT00] N. Medvidovic, R.N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering*, 2000.
- [MYB⁺91] A. MacLean, R. Young, V. Bellotti, T. Moran: "Questions, Options, and Criteria: Elements of a Design Rationale for user interfaces. In *Human Computer Interaction*, vol 6 (3&4), 1991.
- [PCW05] R. Porter, J. O. Coplien, and T. Winn. Sequences as a Basis for Pattern Language Composition. In *Science of Computer Programming*, 56(1-2):231 – 249, 2005.
- [TM08] D. Tamzalit, T. Mens. Guiding Architectural Restructuring through Architectural Styles. In *17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, 2008.
- [UEH⁺01] B. Ulrik, M. Eiglsperger, I. Herman, M. Himsolt, M. Marshall. GraphML Progress Report (Structural Layer Proposal). In *9th International Symposium on Graph Drawing*, 2001.
- [YSJ08] K. Yskout, R. Scandariato, W. Joosen. Change Patterns: Co-evolving Requirements and Architecture. In *Journal of Software and Systems Modeling*, 2008.
- [Zdu07] U. Zdun, Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis. In *Software: Practice and Experience*, 2007.