Abstract — Modern software systems are prone to a continuous evolution under frequently varying requirements. Architecture-centric software evolution (ACSE) enables change in system structure and behavior while maintaining a global view of software to address evolution-centric tradeoffs. Lehman’s law of continuing change demands for long-living and continuously evolving architectures to prolong the productive life and economic value of software systems. To support continuous change, existing solutions for maintenance and evolution fall short of exploiting generic and reusable solutions to address frequent ACSE problems. We argue that architectural evolution process requires an explicit evolution-centric knowledge for pro-active and anticipative change management. We propose a pattern language (PatEvol) as a collection of 7 change patterns that enable reuse-driven and consistent evolution in component-based software architectures (CBSAs). We integrate architecture change mining (PatEvol development) as a complementary and integrated phase to facilitate reuse-driven architecture change execution (PatEvol application). In the proposed pattern language, reuse-knowledge is expressed as a network-of-patterns that build on each other to facilitate a generic, first-class abstraction — that can be operationalised and parameterised - to address recurring evolution tasks. We exploit language-based formalism to promote patterns and prevent potential anti-patterns during ACSE. The pattern language itself continuously evolves with acquisition of new patterns from architecture change logs over-time.

Keywords — Pattern Definition, Pattern Detection, Pattern Language, Software Architecture Evolution, Evolution Reuse

I. INTRODUCTION

Modern software systems operate in a dynamic environment with frequent changes in stakeholders’ needs, business and technical requirements and operating environments [1, 2]. These changing requirements trigger a continuous evolution in deployed software that must be addressed while maintaining a global view of system to effectively resolve evolution-centric tradeoffs [2]. Component-based software architecture (CBSA) represents a system’s structure as topological configurations of components and their interconnections by abstracting implementation-specific details. During evolution the role of CBSA – as a blue-print of evolving system [2] – is pivotal to fill the gap between changing requirements and refactored source code [3].

Lehman’s law of ‘continuing change’ [2] poses a direct challenge for research and practice to support long-living and continuously evolving architectures under varying requirements [3]. The existing solutions promoted the ‘build-once, use-often’ philosophy by exploiting change patterns [3, 4] and evolution styles [5] to address continuous architecture-centric software evolution (ACSE). Recently we conducted a systematic literature reviews (SLRs) to classify and compare the existing research and practices that a) enable architecture evolution [6] and b) facilitate architecture evolution reuse [7]. We observed that a critical challenge to tackle recurring evolution lies with a continuous empirical acquisition of evolution-centric knowledge that could be shared and reused to guide change management [6]. In [7], we define evolution reuse-knowledge as: “[…] a collection and integrated representation (problem-solution mapping) of empirically discovered, generic and repeatable change implementation expertise; that can be shared and reused as a solution to frequent (architecture) evolution problems”.

We observed that reuse-knowledge in existing solution is expressed as patterns [3, 4] and styles [5] to achieve ACSE. However, it is clear that [3, 4, 5] the potential beyond individual patterns and styles could only be maximised with a network-of-patterns that build on each other to enable an incremental and generative evolution in CBSA [6, 7]. We propose a language (PatEvol) as a formalism and collection of change patterns that support reusable solutions to recurring evolution problems. Our solution is inspired by Alexander’s seminal theory [11] about pattern languages that integrate patterns as repeatable solution to build complex architectures. We believe that by exploiting the vocabulary and grammar of a language, individual patterns can be formalised and interconnected to support reusable, off-the-shelf evolution expertise. We identify the central research challenge as:

How to foster an explicit evolution-centric knowledge that enables modeling and executing reuse of frequent evolution tasks in software architectures?

We propose to integrate architecture change mining as a complementary phase for a continuous incremental acquisition of evolution-centric knowledge to enable reuse in change execution. Change mining enables postmortem analysis of architecture evolution histories [8, 9] to i) empirically discover change patterns and ii) derive a pattern language (PatEvol) as a collection of evolution reuse-knowledge. Change execution relies on the network-of-patterns in PatEvol that build on each other to formalise a generic problem-solution view to enable reusable evolution plans. Component-based architecture models and their evolution define the target domain for pattern language. The novelty in the proposed solution lies with:

– An empirical acquisition of evolution-centric knowledge and formalism support to guide ACSE. Pattern language promotes patterns as generic, off-the-shelf expertise for evolution in CBSA.

– A generative and reusable change support for CBSAs as a step towards preventing the potential anti-patterns to enable structural consistency of evolving architecture models.

We summarise related research in Section II and the application domain of pattern language in section III. The proposed solution to promote change patterns for ACSE is presented in Section IV. Pattern language development and its application are detailed in Section V and VI respectively. We discuss change anti-patterns are in Section VII with conclusions and outlook in Section VIII.

II. STATE-OF-THE-ART OVERVIEW

Based on our systematic literature review for classification and comparison of ACSE [6, 7], we identified a lack of efforts to exploit change patterns [4] and styles [5] as the primary artifacts of evolution reuse. The potential for research on patterns and pattern languages to address development and evolution challenges in software life-cycle
are also highlighted with a series of conferences as PLoP\textsuperscript{1} and EuroPLoP\textsuperscript{2}.

A. Reuse-Driven Evolution in Software Architectures

We observed that reuse-knowledge [6, 7] enables an architect to model, analyse and execute recurring changes in continuously evolving software architectures [1]. More specifically, change patterns [3, 4] and evolution styles [5] emerged as the most prominent solutions to achieve reuse for design-time evolution as well as run-time adaptation of architectures. Change patterns follow the conventional philosophy behind the famous Gang-of-Four (GOF) design patterns [10]. However, in contrast to design patterns; change patterns extend the reuse rationale to specifically address evolution-centric issues. Over the last decade, change patterns emerged as an established solution to enable reuse in requirements and architecture co-evolution [3], architecture integration [4] and run-time self-adaptations [1].

It is apparent that [3, 4, 7], pattern-based solution proved effective in systematically addressing the corrective, perfective and adaptive changes to support design-time as well as the run-time evolution in architectures. However, there is a need to formalise the structure and semantic relationships among pattern collections to derive a language support for evolution [10, 11].

B. Pattern Languages for Software Evolution

Evolution is an integral part of modern software life-cycle, however; there is a clear lack of focus to exploit the potential of pattern languages to support a generative and incremental evolution. The only notable work for language-driven architecture evolution is presented in [12]. The authors propose an incremental migration of document archival legacy software to a more flexible architecture using migration patterns. The solution offers a pattern language to formalise the syntax, semantics and pattern relationship for migrating C language implementations to components in an object system. Considering source-code refactoring, authors in [13] introduce a runtime framework for object method transformation.

Identifying the gaps in existing research, we propose change patterns [9, 14] as generic reusable abstractions that could be empirically a) identified, b) specified once and c) instantiated multiple times to specifically benefit evolving CBSAs.

III. RESEARCH CHALLENGES AND PROPOSED SOLUTION

In this section, we outline the core research challenges in terms of an empirical acquisition and application of reuse knowledge to guide ACSE. We also elaborate on the proposed solution that integrates change mining (pattern discovery) as a complementary phase to change execution (pattern-based evolution).

Challenge 1 – Fostering Pattern-Driven Reusable Evolution

A critical challenge [7] in addressing evolution is to empirically discover an explicit evolution-centric knowledge to anticipate change management. The questions then arise, what exactly is reuse knowledge in ACSE? And how to derive and express it?

In our systematic review [6, 7], we provide an evidence-based definition for architecture-centric evolution reuse knowledge (cf. Section I). More specifically, in Figure 1 we propose architecture change mining to empirically derive an explicit reusable knowledge as a pattern language (PatEvol). The pattern language represents a formalised collection of architecture change patterns to map problem-view of the domain (i.e; family of evolving CBSAs). Therefore, a change pattern provides a generic, reusable abstraction to operationalise frequent architecture evolution problems. In contrast to pattern invention in [3, 4], we investigate architecture change logs [9, 14] to empirically discover a classified composition of change patterns and possible variants (i.e: vocabulary) in Figure 1. The technical details about pattern discovery in terms of the complexity, formalism along with algorithmic and tool specific details are presented in our previous work [14, 9]. Pattern model governs the structural composition and semantic relationships among pattern elements (i.e; grammar). The last step involves creating a network-of-patterns in PatEvol. Reuse-knowledge is expressed as a collection of interconnected patterns that build on each other to facilitate a generic and reusable evolution in CBSAs.

Challenge 2 – Enabling Reuse Knowledge-Driven Evolution

In ACSE, an evolution strategy refers to a systematic mapping between the problem-solution views and to derive change implementation mechanism [5, 3]. As oppose to utilising ad-hoc and once-off change execution, the challenge lies with application of reusable evolution strategies to address evolution [4].

In Figure 1, we propose architecture change execution (PatEvol application) as pattern-driven reusable evolution of component-architectures. In the proposed solution, an architect specifies change request (as addition, removal, or modification) of elements in existing CBSA. A declarative specification of change request enables selection of appropriate pattern sequences to derive reusable evolution strategy based on given evolution scenarios. Pattern language provides a method of systematic reuse based on an incremental application of patterns from collection.

During change execution, pattern selection problem refers to selecting the most appropriate pattern from the language collection. We address this with Question-Option-Criteria (QOC) methodology [15] to evaluate the forces and consequences of a given pattern in change execution [10].

IV. APPLICATION DOMAIN FOR PATTERN LANGUAGE

The target domain for PatEvol is family of architectures modeled as component and connectors (i.e; CBSA models). The applicability of the language for other specifications such as object and service-oriented architectures is not guaranteed.
A. Design Model for CBSAs

In CBSA [1], components represent the first class entities as computational elements or data stores of architecture model, illustrated in Figure 2. Every component must contain one or more ports to provide or acquire functionality from other components (CustomerBilling, CustomerPayment in Figure 2). Connectors in CBSA provide an interconnection among two or more architectural components (PaymentData in Figure 2). In addition, configurations represent topological composition of components and connectors in the architecture model (BillingPayment in Figure 2). The components are classified as:

– Atomic Component is the most fundamental type of a component that could not be decomposed. Examples of atomic components are CustomerPayment, CustomerInfo, GetInvoice.

– Composite Component represents a component that contains an internal architecture as a configuration of components and connectors inside composite component. The architecture elements contained in composite components are referred to its child. Example of composite component is CustomerBilling.

\[ V_{PatEvol} = \{ CLS \Rightarrow PAT_{<var_{r_1}, \ldots, var_{r_n}>} \} \ldots (1) \]

\[ G_{PatEvol} = \{ PAT_{<ClassifiedBy> \Rightarrow CLS}, PAT_{<ComposedOf> \Rightarrow OPR}, PAT_{<ConstrainedBy> \Rightarrow CNS}, PAT_{<Evolution> \Rightarrow ARCH} \} \ldots (2) \]

\[ N_{PatEvol} = \{ PAT_{I<var_{r_1}, \ldots, var_{r_n}>}, \ldots, PAT_{N<var_{r_1}, \ldots, var_{r_n}>} \} \ldots (3) \]

A. Change Logs as a Source of Evolution Knowledge

The language vocabulary \((V_{PatEvol})\) is a classified (CLS) composition of change pattern (PAT) instances and their possible variants (VAR) expressed as (1), \(\Rightarrow\) represents a composition operator. To derive vocabulary, we rely on a continuous empirical discovery of change patterns by investigating architecture change logs [12] as a history of sequential change. An Architecture Change Log (ACL) refers to a collection of change instances (CI) that is expressed as: \(ACL = \{CI_1, CI_2, \ldots, CI_N\}\). Currently, ACL contains a sequential collection of change instances for i) an Electronic Bill Presentation and Payment (EBPP) and Tuss Reservation System (TRS) architectures. ACL provides an (extensible) change repository to continuously discover recurring changes. We propose to formalise change instances in the log as an attributed graph (AG) [16] generalised as tuple \(G_i = \{N, E, A_{[N, E]}\}\). Node represents an individual change instance on architecture element; Edge: Edge maintains change sequencing among a node collection while \(A_{[N, E]}\): Attributes on nodes and edges express the intent, scope and impact for change instances explicitly. We exploit sub-graph mining – a formalised knowledge discovery technique to identify the frequent change sequences (i.e, recurring sub-graphs in \(G_i\)) as potential patterns.

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Consequences</th>
<th>Pattern Intent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Mediation ((CI_u, &lt;CI, CM, C2&gt;))</td>
<td>Integrates a component CM  among two or more directly connected components CI, C2</td>
<td></td>
</tr>
<tr>
<td>Functional Slicing ((C, &lt;CI, C2&gt;))</td>
<td>Splits a component C in two or more components CI, C2</td>
<td></td>
</tr>
<tr>
<td>Functional Unification ((&lt;CI, C2&gt;, C))</td>
<td>Merge two or more components CI, C2 into a single component C</td>
<td></td>
</tr>
<tr>
<td>Active Displacement ((C2, CI, &lt;CI, C2&gt;, &lt;CI, C3&gt;))</td>
<td>Replace an existing Component C2 with CI while maintaining connection with existing component CI</td>
<td></td>
</tr>
<tr>
<td>Child Creation ((CI, X1:CI))</td>
<td>Create a child component X1 inside an atomic Component CI</td>
<td></td>
</tr>
<tr>
<td>Child Adoption ((X1:CI, &lt;CI, X1&gt;, &lt;CI, X2&gt;))</td>
<td>Move an internal component X1 from its parent component CI to another component C2</td>
<td></td>
</tr>
<tr>
<td>Child Swap ((X1:CI, X2:CI, &lt;CI, X2&gt;, &lt;CI, X1&gt;))</td>
<td>Swap an internal component X1 in CI to X2 in C2</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. List of Change Patterns Discovered from Logs.

A list of empirically discovered occurrences from change logs are presented in Table 1 that only represents the pattern name, its consequences on architecture model and pattern intent. The Pattern list in Table 1 is not exhaustive. However, our solution allows a continuous change log mining to discover new patterns that also contributes to the evolution of pattern language vocabulary overtime. A graph-based template for pattern specification is detailed in next subsection.
B. Formalising Pattern Language Grammar

The language grammar (PatEvol) is represented as structural composition and semantic relationships among elements in pattern models. We exploit attributed graphs [9, 14] to formalise the language grammar in Listing 1 (cf. Line 02 – 14) as:

Graph-based Template for Pattern Specification Pattern

Graph-based modeling for language grammar is beneficial for Visualising Pattern Composition and Relations to abstract a complex pattern hierarchy. Pattern visualisation facilitates analysing pattern structure to evaluate possible consequences and alternatives. It enables establishing a (graph) Network of Patterns to define semantic relationships among individual patterns.

Nodes – Pattern Composition in Listing 1 is expressed as CLS that represents a classification of patterns (categorising addition, removal, modification types of change impact). OPR represents operationalising changes. CNS represents a set of constraints as pre- and post-conditions to be preserved during pattern-based change. ARCH represents the architecture model to which a pattern is applied (i.e. CBSA). VAR represents the possible variation of a pattern instance. Therefore, a pattern instance PAT is formally expressed as a constrained composition of change operationalisation on CBSA model in Listing 1 (cf. Line 03 – 08).

Edges – Semantic Relations that enable compositional semantics. For example, the possible relation among a pattern and change operators is defined as: PAT — OPR, specifying that pattern is a composition of change operations (cf. Line 09 – 13).

Listing 1. PatEvol Grammar with Graph Modeling Language.

C. Network-of-Patterns

The network-of-patterns represents a collection of interconnected patterns in the language (3), where ⊗ is a language sequencing operation (cf. Line 02 – 15). Pattern selection problem in PatEvol is addressed with a (semi-)formal mapping between the problem-solution views by employing the Question-Option-Criteria (QOC) methodology [15]. Questions refer to an evolution scenario, ii) Options provide the available patterns, while iii) selection depends on Criteria to evaluate forces and consequences [10].

VI. CHANGE EXECUTION – PATEVOl APPLICATION

A. Overview of the Pattern Relationships

Figure 3 presents an overview of the pattern relationships in the language for empirically discovered pattern instances in Table 1. Pattern relationships are of two types:

- Static Relationship: are predefined relations that define specialised and generalised type patterns in the language. For example, uses relation in Figure 3 is a static relation that demonstrates that generic ComponentMediation may use either of the Child Swap/Adoption/Creation type specialised patterns. The relation InverseOf illustrates that pattern instances for Functional Slicing and Component Unification reciprocate each other.

- Dynamic Relationship: In a more practical context static relations are limiting and do not provide details about what is a sequential relation among the patterns in the language. The sequential relation is defined with a keyword follows that represents that if a pattern is dependent or independent on other patterns in the language. During an incremental evolution patterns in the language could dynamically create or remove the relationships to support an incremental change execution.

Figure 3. Overview of the Pattern Language (PatEvol)

B. Language Support for Pattern-based Evolution in CBSA

We illustrate the role of PatEvol to enable an incremental and reusable evolution of architecture with an incremental instantiation of pattern instances. In a partial architecture for EBPP; CustomerBilling and CustomerPayment components are directly connected in Figure 2 (cf. Section IV).

- Change Request: in the existing architecture, selection of a payment type option requires integration of a mediator component (PaymentType) that allows the customer to select the payment type method (debit, credit). Change request by the architect is expressed as QOC formalism [5].

- Pattern-based (Reusable) Evolution Strategy: is illustrated in Figure 4 to support pattern-driven evolution.

1. Question expresses an evolution problem: How to integrate a mediator CustomerPayment (C_M) among directly connected components (C_1, C_2)? The architect species this as: Integrate(PRE, (C_1, CM, C_2), POST)

2. Option provides pattern selection: from pattern collection in language. This is expressed as: Select a pattern instance that eliminates the direct interconnection between CustomerBilling and CustomerPayment and mediate them with PaymentType.

3. Criteria define pattern consequences: preconditions (PRE) represent the architecture model before change. The post-conditions (POST) represent the evolved architecture model as a consequence of applying the ComponentMediation pattern.

In Figure 4, the ComponentMediation pattern (cf. Table 2), addresses the evolution problem to introduce a broker between billing and payment. In addition, we observed with some variations ComponentMediation could also be reused for:

- Extending a peer to peer (P1, P2) interconnection in the architecture with a server (S) component: Integrate (P1, S, P2)

- Introduction of an authentication service (A) between billing and Payment (B, P) as Integrate (B, A, P), etc.
VII. IDENTIFIED ANTI-PATTERNS AND DISCUSSION

A. Anti-pattern Identification and Prevention

The role of pattern language is central in promoting patterns to achieve reuse and consistency in evolution for CBSA. However, change patterns do not guarantee an optimal solution to a given evolution problem, instead they support an alternative and reusable solution. Structural and semantic consistency of CBSA [1] may be violated as a consequence of pattern-based evolution. These counter-productive and negative impacts of change patterns on architecture models arise in change anti-patterns. It is vital to identify these anti-patterns and prevent them from ensuring CBSA model consistency. We discuss two identified anti-patterns and possible prevention in Figure 5.

**Anti-pattern I – Orphan Child** - As illustrated in Figure 4 a), orphan child is an anti-pattern as a consequence of removing the composite component having an internal architecture. This means removal of the composite parent component creates orphan children (internal sub-component: Childs and Childd). An orphan component cannot coordinate with others in architecture.

**Prevention 1** – Whenever a parent is removed, all of its children must be removed (i.e; kill the child if parent is dead).

**Prevention 2** – Apply the Child Adoption pattern to accommodate an orphan (i.e; adopt the child if parent is dead).

**Anti-pattern II – Bulky Component** - As illustrated in Figure 4 b), bulky component is an anti-pattern that results as a consequence of component composition that has a complex and monolithic internal architecture. Composite component (Parent) comprises a bulk of child components (A, B, C) in the Figure 4 b).

**Prevention** – Apply Functional Slicing pattern to split the parent into fine-grained and specialised components, Figure 4b).

B. Tool Support and Preliminary Evaluations

We provide a tool chain for pattern identification and pattern application. In [14], we detail about graph-based formalism and prototype G-Pride (Graph-based Pattern Identification) to discover patterns from architecture change logs. We provide a tool G-Pride (Graph-based Pattern Identification) to discover change patterns from logs in an automated fashion. In addition, we provide PatEvol [17] (Pattern-based Evolution) to specify the change intent, retrieve and instantiate change patterns.

**Preliminary evaluations** are primarily focused on assessing the adequacy of patterns in the language to address recurring evolution problems in CBSA [3, 4, 1]. Moreover, the case studies (cf. Section III) and the data generated from the PatEvol prototype provide the early validations of this solution. Further evaluations of the proposed solution are required in a more practical context.

VIII. CONCLUSIONS AND OUTLOOK

We argue about the role of pattern language as an explicit knowledge-base to support reuse-driven evolution in CBSAs. The solution promotes language as a formalised collection of patterns that enable reuse and consistency in ACSE. We discuss about the negative consequences of patterns that affect architecture consistency as change anti-patterns.

In future, we primarily focus on classification of change patterns as commutative and dependent patterns. We analyse the extent to which architecture evolution could be parallelised (i.e.; identifying dependent and independent change patterns).

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


