PatEvol – A Pattern Language for Evolution in Component-Based Software Architectures

Aakash Ahmad, Pooyan Jamshid, Claus Pahl, Fawad Khaliq
School of Computing, Dublin City University, Ireland
[ahmad.aakash || pooyan.jamshidi || claus.pahl || fawad.khaliq]@computing.dcu.ie

***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 a continuous change, the existing solutions 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 architecture evolution. 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 to operationalise 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 an incremental 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 a frequent change in stakeholders' needs, business and technical requirements and operating environments [1, 2]. These changing requirements trigger a continuous evolution in deployed software that needs to 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 practices that aim 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 a continuous architecture-centric software evolution (**ACSE**). Recently we conducted 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; 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. 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 and cities. 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 identified 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:

– 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.

– Generative and reusable change support for CBSA. Language-based formalism is a step towards preventing the potential anti-patterns to enable structural consistency of architecture model.

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[1] and EuroPLoP[2].

A. Reuse-Driven Evolution in Software Architectures

Reuse-knowledge empowers the role of an architect to model, analyse and execute recurring changes in continuously evolving software architectures [1]. We observed, *change patterns* [3, 4], *evolution styles* [5] as the most prominent solutions to achieve reuse for design-time evolution as well as run-time adaptation. 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].

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 semantics 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 while addressing frequent evolution is to acquire an explicit evolution-centric knowledge for anticipative change management. The questions then arise, *what exactly is reuse knowledge in ACSE*? And *how to derive and express it*?

[1] PLoP - Conference on Pattern Languages of Programs: www.hillside.net

[2] EuroPLoP - European Conference on Pattern Languages of Programs

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-solution 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. 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 pattern-based anticipated evolution.

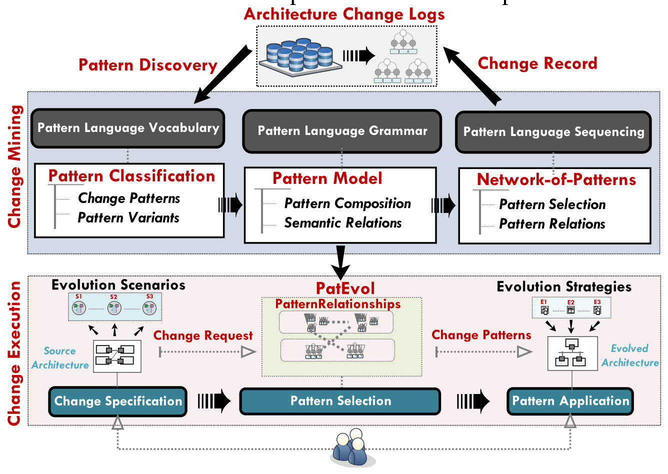


Figure 1. Layered Overview of Proposed Solution – *Change Mining* (PatEvol Development) and *Change Execution* (PatEvol Application).

Challenge 2 – Enabling Reuse Knowledge-Driven Evolution

In ACSE, an evolution strategy refers to a systematic mapping between the problem-solution views 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. We summarise the architecture model and running example of evolution in CBSA as follows.

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.

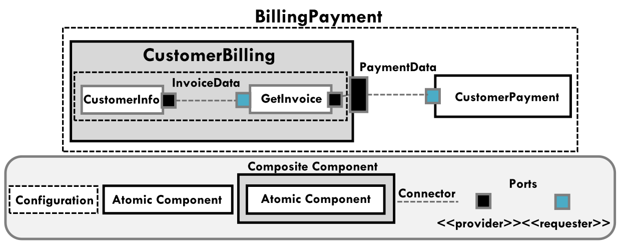


Figure 2. Partial Architecture View for EBPP System.

B. Evolution in CBSAs

Figure 2 represents a partial architectural view for an *Electronic Billing Presentment and Payment* (EBPP) system, also used as a running example. It represents a co-ordination among the CustomerBilling and CustomerPayment components to pay customer bills. The existing architecture (partial-view) in Figure 2 has the following evolution scenario.

Integration of a PaymentType Component – The existing architecture only supports a direct debit mechanism for bill payments by customer. The evolved architecture must enable the customers to select the type of payment as either direct debit or card-based credit payment. In an architectural context of Figure 2, this could be achieved with integration of a mediator component PaymentType among directly connected components CustomerBilling and CustomerPayment.

V. Change Mining – PatEvol Development

In the proposed solution, change mining (cf. Figure 1) aims at an empirical analysis of change logs to investigate architecture evolution history. The language is composed of i) a classified composition of patterns and their variants (1. vocabulary: $\mathrm{V}_{\boldsymbol{PatEvol}}$) along with a ii) set of rules that govern the structure and semantic relations among pattern elements (2. grammar: $\mathrm{G}_{\boldsymbol{PatEvol}}$) to create a iii) network-of-patterns (3. sequencing: $\mathrm{N}_{\boldsymbol{PatEvol}}$). We propose to exploit an attributed graph-based formalism [14, 9, 16] to i) discover change patterns and ii) formalise the structural and semantic relationships of pattern language as: $PatEvol$ ($\mathrm{V}_{PatEvol}$ x $\mathrm{G}_{PatEvol}$ x $\mathrm{N}_{PatEvol}$) =

$$\left[\begin{array}{c} \mathrm{V}_{\boldsymbol{PatEvol}} = \{CLS \ \rhd PAT_{<var_1,\dots,\, var_n>}\} \dots (1) \\ \\ \mathrm{G}_{\boldsymbol{PatEvol}} = \{PAT_{<ClassifiedBy>}CLS, \\ PAT_{<ComposedOf>}OPR, \\ PAT_{<ConstrainedBy>}CNS, \\ PAT_{<Evolves>}\ ARCH\} \dots (2) \\ \\ \mathrm{N}_{\boldsymbol{PatEvol}} = \{PAT_{1<var_1,\dots,\, var_n>} \bowtie \dots \bowtie PAT_{n<var_1,\dots,\, var_n>}\} \dots (3) \end{array}\right]$$

A. Change Logs as a Source of Evolution Knowledge

The ***language vocabulary*** ($\mathrm{V}_{\boldsymbol{PatEvol}}$) is a classified (*CLS*) composition of change pattern (*PAT*) instances and their possible variants (*VAR*) expressed as (1), $\rhd$ 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: $ACL\ = <CI_1,\ CI_{2\dots},CI_N\ >$. Currently, ACL contains a sequential collection of change instances for i) an *Electronic Bill Presentment and Payment* (EBPP) and *Tour 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 $G_c\ = <N,\ E,\ A[_{N,\ E}]>$, **N : Node** represents an individual change instance on architecture element; **E : 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_c) as potential patterns.

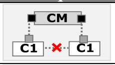
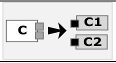
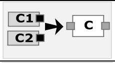
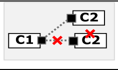
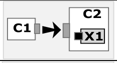
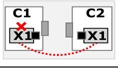
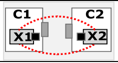
Pattern Name	Consequences	Pattern Intent
Component Mediation ***(C_M, <C1, CM, C2>)***		Integrates a component **CM** among two or more directly connected components **C1**, **C2**
Functional Slicing ***(C, <C1, C2>)***		Splits a component **C** in two or more components **C1**, **C2**
Functional Unification ***(<C1, C2>, C)***		Merge two or more components **C1**, **C2** into a single component **C**.
Active Displacement ***(C2:C3, <C1, C2>, <C1, C3>)***		Replace an existing Component **C1** with **C3** while maintaining connection with existing component **C2**
Child Creation ***(C1, x:C1)***		Create a child component **x1** inside an atomic Component **C1**
Child Adoption ***(x:C1, <C1,x>, <C2, x>)***		Move an internal component **x** from its parent component **C1** to another component **C2**
Child Swap ***(x:C1, y:C2, <C1,x>, <C2, y>)***		Swap an internal component **x** in **C1** to y in **C2**

Table 1. List of Change Patterns Discovered from Logs.

A list of empirically discovered patterns 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*** ($G_{PatEvol}$) is represented as structural composition and semantic relationships among elements in pattern model.We exploit attributed graphs [9, 14] to formalise the language grammar in Listing 1 (cf. Line 02 – 04) 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 $\xleftarrow{composedOf}$ OPR, specifying that pattern is a composition of change operations (cf. Line 09 – 13).

```
01: <graphml>
02:  <graph id = "PAT_N" edgedefault = "directed">
03:    <node id = "CLS"> ...</node>
04:    <node id = "PAT"> ...</node>
05:    <node id = "OPR"> ...</node>
06:    <node id = "CNS"> ...</node>
07:    <node id = "ARCH">... </node>
08:    <node id = "VAR"> ...</node>
09:    <edge id = "Classifies" source="CLS" target="PAT"/>
10:    <edge id = "ComposedOf" source="PAT" target="OPR"/>
11:    <edge id = "ConstrainedBy" source="PAT" target="CNS"/>
12:    <edge id = "Evolves" source="PAT" target="ARCH"/>
13:    <edge id = "HasVariation" source="PAT" target="VAR"/>
14:  </graph>
15: </graphml>
```

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 Component Mediation 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.

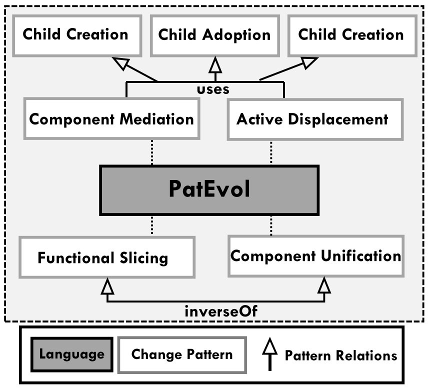


Figure 3. Overview of the Pattern Language (PatEvol)

B. Language Support for Pattern-based Evolution in CBSA

We illustrate the role of PatEvol to enable a generative 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, (C1, CM, C2), 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* Customer Payment *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.*

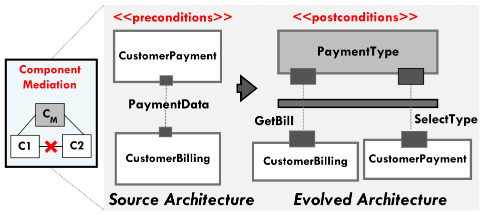


Figure 4. Pattern-based Evolution for Component Integration.

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 pattern 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] model may be violated as a consequence of pattern-based evolution. These *counter-productive and negative impacts of change patterns on architecture model results in change anti-patterns*. It is vital to identify these anti-patterns and possibly prevent them to ensure 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: Child$_X$ and Child$_Y$). An orphan component cannot co-ordinate 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).

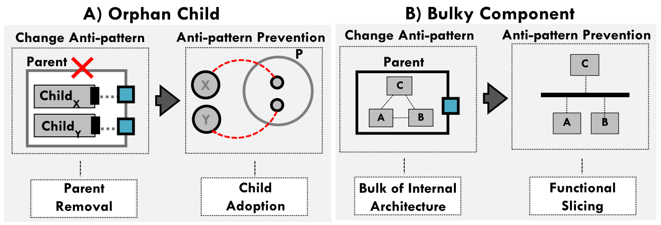


Figure 5. An Overview of Preventing Change Anti-patterns.

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 pattern from logs in an automated fashion. In addition, we provide PatEvol [14] (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 see a 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, but negative consequences of patterns can 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).

References

[1] T. Mens, Tom, J. Magee, and B. Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer, 2010.*

[2] M. Lehman, Laws of Software Evolution Revisited. *In Software Process Technology, LNCS 1996.*

[3] Y. Koen, R. Scandariato, and W. Joosen. Change Patterns: Co-evolving Requirements and Architecture. In *SoSyM , 2012*.

[4] I. Côté, M. Heisel, and I. Wentzlaff. Pattern-Based Evolution of Software Architectures. In *ECSA, 2007*.

[5] J. Barnes, D. Garlan and B. Schmerl. Evolution Styles: Foundations and Models for Software Architecture Evolution. In *SoSyM , 2012*.

[6] P. Jamshidi, M. Ghafari, A. Ahmad and C. Pahl. A Framework for Classifying and Comparing Architecture Centric Software Evolution. In *CSMR 2013*.

[7] C. Pahl, S. Giesecke and W. Hasselbring. An Ontology-based Approach for Modelling Architectural Styles. European Conference on Software Architecture ECSA07. 2007.

[8] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE, 2004*.

[9] A. Ahmad, P. Jamshidi, Claus Pahl. Graph-based Implicit Knowledge Discovery form Architecture Change Logs. In *SHARK 2012*.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design patterns: Abstraction and Reuse of Object-oriented Design. *In ECOOP 1993*.

[11] C. Alexander. The Origins of Pattern Theory, The Future of the Theory, and the Generation of a Living World. *IEEE Software*, 1999.

[12] M. Goedicke and U. Zdun. Piecemeal Migration of a Document archive System with an Architectural Pattern Language. In *CSMR 2001*.

[13] U. Zdun. Supporting Incremental and Experimental Software Evolution by Runtime Method Transformations. In *JSCP, 2004*.

[14] A. Ahmad, P. Jamshidi, and C. Pahl. Graph-based Pattern Identification from Architecture Change Logs. In *IWSSA 2012*.

[15] A. MacLean, R. Young, V. Bellotti, T. Moran: "Questions, Options, and Criteria: Elements of a Design Rationale for user interfaces, *Human Computer Interaction*, vol 6 (3&4), 1991.

[16] H. Ehrig, U. Prange, G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In *Graph Transformations*, 2004.

[17] M.X. Wang, K.Y. Bandara and C. Pahl. Integrated constraint violation handling for dynamic service composition. IEEE Intl Conf on Services Computing. 2009.