Exploiting Patterns and Tool Support for Reusable and Automated Change Support for Software Architectures

Fawad Khaliq(1), Aakash Ahmad(2), Onaiza Maqbool(3), Pooyan Jamshidi(4), and Claus Pahl(5)

(1) Department of Computer Science.Quaid-i-Azam University (Pakistan)
E-mail: fawad.khaliq@qau.edu.pk

(2) School of Electrical Engineering and Computer Science.National University of Sciences and Technology (Pakistan)
E-mail: aakash.ahmad@seecs.edu.pk

(3) Department of Computer Science.Quaid-i-Azam University (Pakistan)
E-mail: onaiza@qau.edu.pk

(4) Irish Centre for Cloud Computing and Commerce.Dublin City University (Ireland)
E-mail: pooyan.jamshidi@computing.dcu.ie

(5) Irish Centre for Cloud Computing and Commerce.Dublin City University (Ireland)
E-mail: claus.pahl@computing.dcu.ie

ABSTRACT

Lehman's law of *continuing change* implies that software must continually evolve to accommodate frequently changing requirements in existing systems. Also, maintainability as an attribute of system quality requires that changes are to be systematically implemented in existing software throughout its lifecycle. To support a continuous software evolution, the primary challenges include (i) enhancing reuse of recurring changes; and (ii) decreasing the efforts for change implementation. We propose change patterns and demonstrate their applicability as reusable solutions to recurring problems of architectural change implementation. Tool support can empower the role of a designer/architect by facilitating them to avoid labourious tasks and executing complex and large number of changes in an automated way. Recently, change patterns as well as tool support have been exploited for architecture evolution, however; there is no research to unify pattern-driven (reusable) and tool-supported (automated) evolution that is the contribution of this paper. By exploiting patterns with tool support we demonstrate the evolution of a peer-to-peer system towards client-server architecture. Evaluation results suggest that: (i) patterns promote reuse but lack fine-granular change implementation, and (ii) tool supports automation but user intervention is required to customise architecture change management.

Keywords: Automated Evolution, Change Patterns, Software Architecture, Software Evolution.

1- INTRODUCTION

Evolution of software systems is a continuous phenomenon as a consequence of changes in system requirements and operational environments [1]. Lehman's law of continuing change states that an E-type system that is any real-world software must be continually adapted or it becomes progressively less satisfactory [2]. In addition, maintainability; i.e., the effectiveness with which changes can be implemented to support evolution represents an attribute of software quality as per the ISO/IEC 25010:2011 quality model [3]. Therefore, to accommodate frequently changing requirements; continuous evolution is required to ensure quality and evolvability of existing software [4]. However, to support continuous evolution; an important decision lies in selection of an appropriate software representation such as code vs design vs configurations for change implementation.

Architecture[1] represents the blueprint of a software system by abstracting low-level and implementation specific details. For example, the source-code modules and their interactions can be abstracted with higher-level representations such as architectural components and their connectors [5, 6]. Source code changes in general proved effective for software refactoring as corrective type changes, while architectural evolution primarily aims at perfective and adaptive changes[2] [5]. Moreover, the component-connector architectural representations provide the stakeholders, designers, and developers with a system overview, known as global structure for systematic modelling, development, and evolution of software [6]. Once the decision is reached to exploit the architecture as the driver for software evolution, we are faced with two distinct challenges on (i) how to capitalise existing knowledge and expertise that can be reused to tackle recurring evolution [7], and (ii) how to automate and customise the application of such reuse knowledge [8] – both introduced below.

The concept of applying reuse-knowledge or best practices to design and develop software emerged from the Gang-Of-Four (GOF) design patterns [9]. After more than two decades of research and practice, design patterns have now matured to establish various catalogues and languages as an integral part of software development processes. In contrast, the recent concepts of evolution styles or change patterns as the artifacts of evolution-specific knowledge are innovative but relatively immature [6, 19, 11]. Some industrial [13, 14] as well as academic studies [4, 7, 12] on software evolution have highlighted the growing needs to develop processes, frameworks and patterns that exploit reuse-knowledge to support design and architectural evolution. Specifically, a change pattern packages together changes, constraints, and architectural descriptions as evolution-knowledge that can be reused during change implementation [11, 26].

[1] ISO/IEC/IEEE 42010 *Systems and software engineering is a standard for Architecture description of software systems.*

[2] In literature [4, 5] the terms evolution and change are virtually synonymous and often used interchangeably. However, a distinction must be maintained – a collection of changes on architectural/software description cause their evolution

Automation of change implementation enables the execution of complex tasks such as frequent addition or removal of a significant number of component and connectors that may prove error-prone and time-consuming with manual efforts [8]. In addition, customisation allows designers/architects to incorporate their decisions such as specifying the evolvable and preservable structures or selecting/discarding specific changes on software architecture. Tool support with necessary intervention and customisation enables importing the source and exporting the evolved architecture models and support necessary steps that enable the evolution of source model [8].

In this paper, we aim to unify reusability and automation of changes in software architecture evolution process. The existing research lacks any solutions that integrates change patterns with necessary tool support during architectural evolution [6, 8]. The state of the art on architectural evolution research highlights the needs for solution(s) that supports an empirical discovery of change patterns, known as *reuse knowledge acquisition* and systematic application of discovered patterns to evolve software architectures, known as *reuse knowledge application* [4, 7, 12]. Therefore, considering change patterns as artifacts of evolution specific knowledge; our proposed contributions focuses on:

- Exploiting change patterns as artifacts of evolution specific knowledge with tool support to facilitate reusable and automated change management for software architectures.
- Enabling user intervention in the evolution process to customise and guide architectural change management.

The remainder of this paper is organised as follows. Section 2 provides research challenges and overview of the solution. Section 3 presents a meta-model of pattern-driven and tool-supported architectural change management and introduces the change pattern catalogue. Section 4 demonstrates pattern-based reuse and prototype support for architecture evolution. Section 5 presents evaluations, lessons learnt and validity threats. Section 6 overviews related research. Section 7 concludes the paper.

2- PRESEARCH CHALLENGES AND SOLUTION OVERVIEW

In this section, first we discuss the primary research challenges (in Section 2-1) that follows an overview of the proposed solution that address the challenges (in Section 2-2).

2-1 CHALLENGES

Based on the progress of research on software architectural evolution, we have identified two primary challenges as below.

Challenge I - Integrating Reuse Knowledge and Tool Support in Architecture Evolution Process

Reuse knowledge in terms of change patterns allows the designers and architects to leverage the existing expertise and strategies to tackle recurring problems during architectural evolution. Recently, patterns as artifacts of reuse [19] or tool support to enable automation [22] have been exploited for architecture evolution, however; there is no research to unify pattern-driven and tool-supported architectural evolution. The lack of tool integration in pattern-based architectural evolution process is mainly due to economic and technical challenges [21, 22]. Specifically, the evolution of industrial scale software architecture is driven by timely and economically efficient change implementation that often results in automated but ad-hoc and once-off solutions [19]. Moreover, complexities like continuous pattern discovery, appropriate pattern selection, pattern evaluation and its application during architecture evolution process poses a variety of challanges for developing a comprehensive tool support for pattern-based architectural evolution [8].

Challenge II - Customisation and User Intervention during Architecture Evolution

Another issue in architectural evolution is the customisation of the architectural evolution process as per the requirements specification of architectural elements and proposed changes on them. There is a need to facilitate the designers/architects with incorporation of their decisions, preferences and expertise to guide the architecture evolution process. In this context, a complete automation is desirable, however; both evolution and architecting are intellectual processes that require human intervention and collective decision rather than pure automation of change execution [4, 8].

In addition to the challenges above, it is vital to mention another issue that lies with a continuous discovery of architecture evolution knowledge that must be frequently integrated in the architectural evolution process. The challenges and solution for a continuous discovery of architectural evolution knowledge have been discussed in our previous research [25]. In this paper, we outline the primary research problem as:

How to unify patterns and tool support that incorporates user intervention in the evolution process to enable reusable and automated architectural change management.

2-2 OVERVIEW OF PROPOSED SOLUTION

In order to address the outlined challenges above, we provide an overview of the proposed solution as illustrated in Figure 1. In contrast to the existing research on evolution styles [6] and change patterns [19], in the proposed solution we exploit the existing change patterns from [11, 25] for reusable and automated evolution of software architectures. Considering the context of Figure 1 as the proposed solution, the pattern collection (*P1, P2, P3*) supports multiple changes (*C1, C2*) as addition or removal of architectural components and connectors causing an incremental evolution of architecture models (A1, A2, A3). A collection of tool elements (*T1, T2, T3*) enable the necessary automation. The role of an architect or designer is empowered by selecting the patterns and applying the tool support to enable incremental changes to evolve architectures in a reusable and automated fashion.

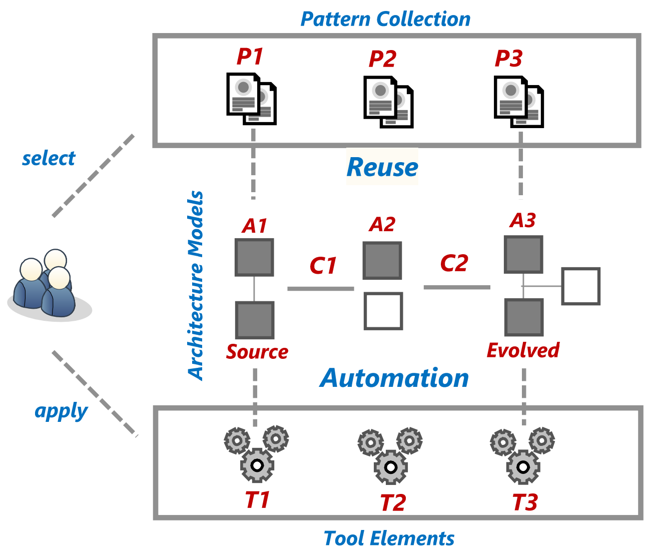


Figure 1 Overview of Pattern-Driven and Tool-Supported Architectural Changes.

Assumptions and Contributions of Solution: This research provides a significant extension to our previous work on automated empirical discovery of patterns by mining architecture change logs [11, 25]. In the present work, we assume the existence of change pattern collection such as pattern languages or pattern catalogues [15, 19]. The contribution of this research is to exploit existing change patterns [15, 25] to automate and customise change support for software architectures. We demonstrate pattern-based reuse and tool-supported automation by evolving a peer-to-peer appointment system towards client-server architecture. In addition, by supporting different parameters such as specification of architectural constraints or change operations, the tool facilitates necessary user intervention to customise architectural changes. The lack of comprehensive case studies from industrial-scale systems for evaluation represents a validity threat to the solution. As part of our ongoing research, acquisition of industrial data for evaluation defines futuristic research.

3- MODELING PATTERN BASED AND TOOL SUPPORTED EVOLUTION

Modeling is the first step towards supporting evolution as it identifies and represents the necessary elements, their composition and interrelations.

Figure 2 provides a meta-model for pattern-based and tool-supported architectural evolution. Based on Figure 2, we discuss the architecture model in Section 3-1 with details of the pattern model in Section 3-2 and evolution tool in Section 3-3. Architecture model and change pattern as a concept as well as implementation remain independent of any specific tool or platform. In the context of Figure 2, any specific tool or technology could only use architecture and pattern model by importing/exporting them but could not change their representation. The elements and concepts presented in this section are utilised throughout the paper.

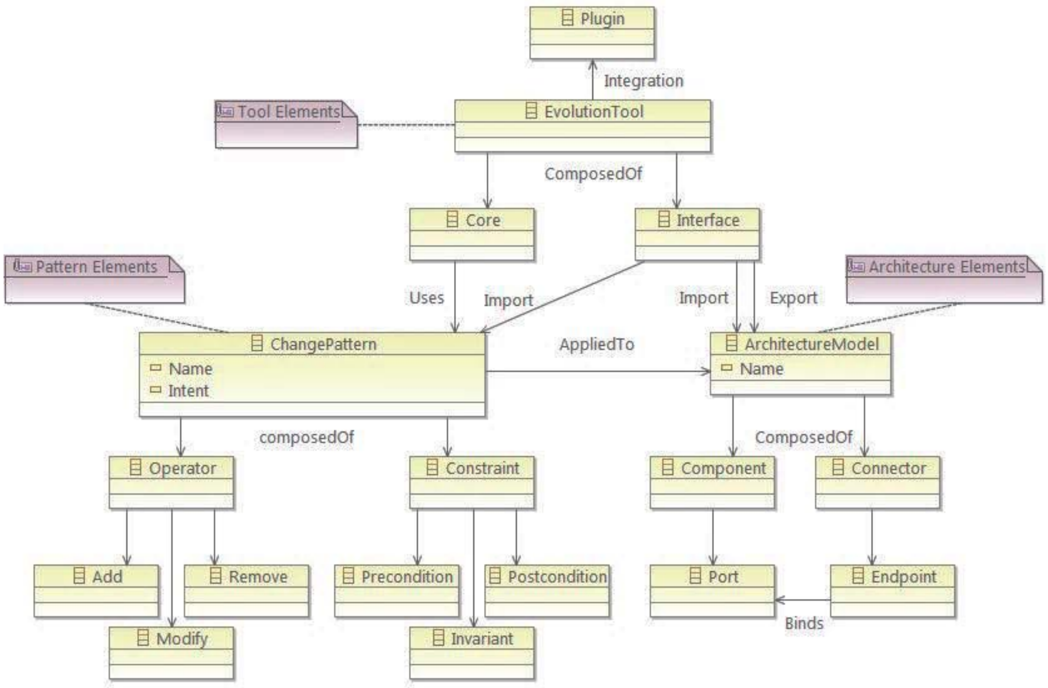
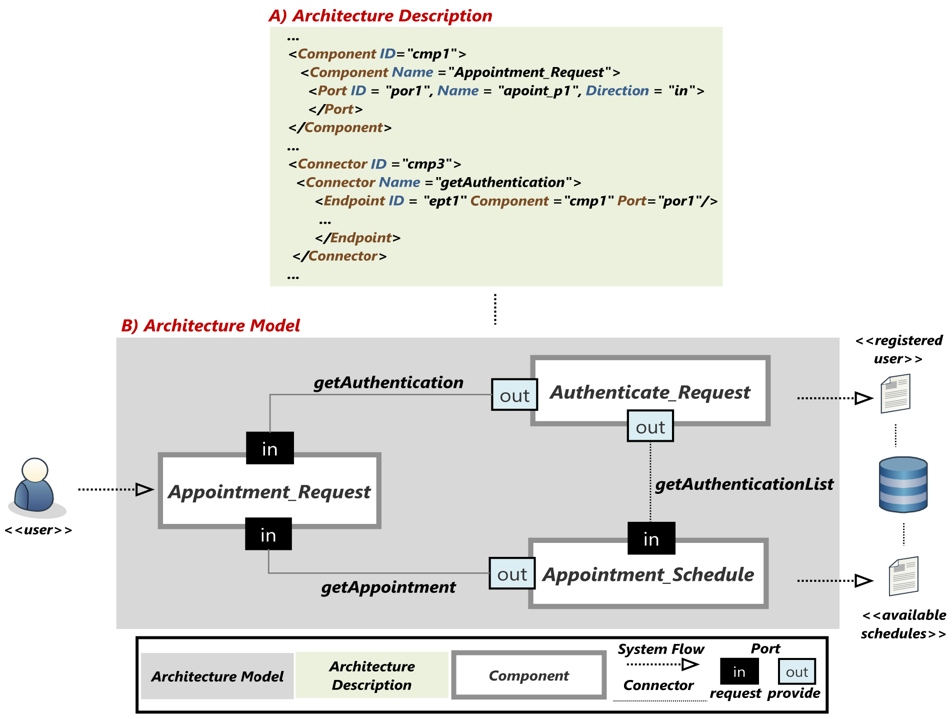


Figure 2 Model Elements and their Interrelationships.

3-1 ARCHITECTURE MODEL

It provides architectural specification in terms of the individual elements that are composed and interconnected to represent the overall structure of software. In Figure 2, the architectural model is specified with architectural elements named *Components* as computational elements of system along with their *Connectors* for component interconnections. This description is based on the well-known view of the model, the so called component and connector (C&C) architectural view [16]. Furthermore, components are composed of *Ports* that either provide (a.k.a. source port) or require (a.k.a. sink port) some functionality – a component's point-of-interaction. In contrast, a connector is composed of *Endpoint* that provides the binding among the provider and requester ports. As modeled in Figure 2, by abstracting the individual changes in patterns, the operations on architectural model are constrained to preserve composition of elements. For example, constraints ensure that whenever a component is added, its corresponding port must also be added to ensure integrity of architectural composition. Based on the model from Figure 2, we discuss a concrete example of the architecture model in Figure 3 that is also used as a running example.



```
...
<Component ID="cmp1">
  <Component Name ="Appointment_Request">
    <Port ID = "por1", Name = "apoint_p1", Direction = "in">
    </Port>
</Component>
...
<Connector ID ="cmp3">
 <Connector Name ="getAuthentication">
     <Endpoint ID = "ept1" Component ="cmp1" Port="por1"/>
      ...
     </Endpoint>
 </Connector>
...
```

Figure 3 Component and Connector Architectural View for P2P-AS System.

Running Example - Architectural view of Peer-to-Peer Appointment System (P2P-AP) presents architectural description as a partial view in Figure 3 A) as well as architecture model in Figure 3 B). Specifically, Figure 3 B) represents three distinct components (**Appointment_Request**, **Appointment_Schedule**, **Authenticate_Request**) and three connectors (**getAppointment**, **getAuthentication**, **getAuthenticationList**) with appropriate ports and endpoints respectively. For space reasons, we abstract system-level details such as user interfacing and data retrieval. We primarily focus on the C&C architectural view as Figure 3 B). **Appointments_Request** gets appointments from **Appointment_Schedule** component. Both **Appointment_Request** and **Appointment_Schedule** components require **Authenticate_Request** component for authenticating requests before issuing appointment schedules. We utilise the P2P-AP architectural representation from Figure 3 as a source model and demonstrate pattern and tool support for its evolution in subsequent sections.

The representation in Figure 3 B) as *architecture model* presents a graphical view of the interconnected components, while an *architectural description* in Figure 3 A) also detailed later provides a suitable notation to formally specify the architecture. It is vital to distinguish between, (i) a model that is a graphical view for human comprehension such as designer/architect, while (ii) description represents a suitable/formalised format for machine representation or tool-based manipulation.

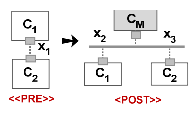
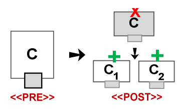
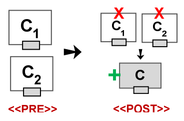
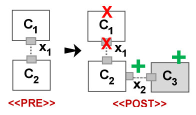
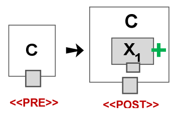
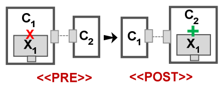
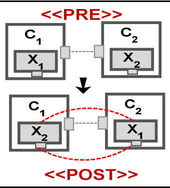
3-2 CHANGE PATTERN

It is composed of *Operators* that specify changes as addition, removal, and modification of architectural components and connectors. It is vital to mention that in the evolution model (cf. Figure 2) there is no direct relation between operators and architecture elements as operators cannot be directly applied to architecture. In traditional change implementation techniques, when individual operators are applied to an architecture model this is referred to as *change primitives* [5]. In contrast to primitives, change patterns abstract the lower-level and ad-hoc changes into generic and reusable changes. Moreover, the constraints specify a set of conditions as precondition, invariant, post-conditions that must be enforced through a pattern before, during and after change implementation. Constraints are vital to preserve certain conditions or architectural structure during and after change execution such as a component must contain one or more port. Based on the elements in Figure 2, a change pattern is modeled as: *a constrained composition of change operationalisation on the architecture model*. A change pattern is a generic and repeatable solution to frequently occurring architecture evolution problems. Examples of change patterns are provided in Table 1. The relation between architecture model and change pattern is expressed as: *Change Pattern* is **AppliedTo** *Architecture Model*, so that pattern-based architecture evolution can be supported.

A) A Catalogue of Architecture Change Patterns

The catalogue represents a collection of patterns ready for selection and reuse. We followed the guidelines in [17] to develop a pattern template presented in Table 1. The seven patterns presented have been discovered by investigating architecture change logs as histories of architecture evolution [11, 25]. By investigating logs, we discovered recurring changes with specific properties as potentially reusable patterns with additional details in [25]. In Table 1, we only provide the necessary elements of the template with extended details of an individual pattern in Table 2. Pattern elements include: (1) *Pattern Name* that provides unique name for each pattern; (2) *Intent* describes the motivation or the known uses, (3) *Operators* represent the required changes to implement the pattern, and finally (4) Pattern overview represents the preconditions and post-conditions of evolution as a reference diagram called *pattern overview* or *pattern thumbnail*.

Table 1 A Catalogue of Change Patterns for Architecture Evolution Aadopted from [25].

Pattern Name	Intent	Operators	Pattern Overview
Component = *CMP*, Connector = *CON*, Add() = Addition Operation, Rem() = Removal Operation, C = Component Instance, X = Connector Instance			
Component Mediation	Component Mediation integrates a mediator component (CM) among two or more directly connected components (C1, C2).	- Add(CM:CMP) - Add(X2 (CM, C1): CON) - Add(X3 (CM, X3): CON) - Rem(X1 (C1, C2):CON)	 C_1 X_1 C_2 <<PRE>> C_M X_2 X_3 C_1 C_2 <<POST>>
Functional Slicing	Functional Slicing split a component (C) into two or more components (C1, C2) for a functional decomposition of C.	- Add(C1: CMP) - Add(C2:CMP) - Rem(C:CMP)	
Functional Unification	Functional Unification merges two or more components (C1, C2) into a single component (C) for a functional unification of (C1, C2).	- Rem(C1: CMP) - Rem(C2: CMP) - Add(C: CMP)	
Active Displacement	Active Displacement replaces an existing component (C2) with a new component (C3) while maintaining the interconnection with existing components (C1, C2).	- Add(C3:CMP) - Rem(C2:CMP) - Add(X2 (C2, C3): CON) - Rem(X1 (C2, C1):CON)	 C_1 X_1 C_2 <<PRE>> C_1 X_1 C_2 X_2 C_3 <<POST>>
Child Creation	Child Creation creates a child component (X1) inside an atomic component (C1), such that C1 is a composite component now.	- Add(X1: CMP) - Mov(C (X1): CMP)	
Child Adoption	Child Adoption adopts a child component (X1) from a composite component (C1) to an atomic component (C2).	- Rem(C1 (X1): CMP) - Add(C2 (X1): CMP)	
Child Swapping	Child Swap enables the swapping of the child components (X1, X2) between composite components (X1, X2).	- Rem(C1 (X1): CMP) - Add(C2 (X1): CMP) - Rem(C2 (X2): CMP) - Add(C1 (X2): CMP)	 <<PRE>> C_1 X_1 C_2 X_2 C_1 X_2 C_2 X_1 <<POST>>

B) Change Primitive vs Change Pattern

After presenting the pattern model (cf. Figure 2) and examples of change patterns (cf. Table 1), we now distinguish between change primitives and change patterns. A primitive change ($\Delta_{primitive}$) is the most fundamental unit of architecture evolution that supports the addition, removal and modification of individual components and their ports and connectors with their endpoints [5]. Change primitive are expressed as below:

$$\begin{aligned}\Delta_{primitive} := \{ & \\ & (\text{Add}_{component}, \text{Add}_{connector}, \text{Add}_{port}, \text{Add}_{endpoint}), \\ & (\text{Remove}_{component}, \text{Remove}_{connector}, \text{Remove}_{port}, \text{Remove}_{endpoint}), \\ & (\text{Modify}_{component}, \text{Modify}_{connector}, \text{Modify}_{port}, \text{Modify}_{endpoint}), \\ \} & \end{aligned}$$

For example, in Table 1 (cf. *ComponentMediation* - pattern 1), the addition ($\text{Add}_{component}$) of an individual component, as CM that is of type CMP is expressed as change primitive: Add(CM : CMP). However, change primitives have following limitations:

- *Explicit Enforcement of Individual Constraints*: unlike pattern-based change representation in Figure 2, a primitive change is directly applied to the architecture elements and it does not enforce any constraints such as component must contain a port. The architects/designers must rely on their knowledge about hierarchy of elements to manually add a port after component addition to maintain architectural composition.
- *Limited Reuse of Change*: a primitive change offers a limited reuse with addition or removal of individual components and connectors. For example, considering *ComponentMediation* pattern above; when there is a change affecting multiple components and connectors such as component integration each of the individual changes must be explicitly specified. This can prove labourious and error-prone when the total number of changes to be implemented are large and complex.

In contrast to change primitives, as modeled in Figure 2 and illustrated in Table 1 a change pattern abstracts change primitives and constraints to enable composite changes such as integration, composition, replacement of a collection of components and connectors. We further discuss the benefits of a pattern-based change in the context of a scenario in Table 2. In Table 2, the instance of *Active Displacement* pattern (cf. Table 1) is provided with details of pattern application on P2P-AS architecture (cf. Figure 3). For pattern examples in Table 2, we use the terminologies *atomic component* that is a component with no internal sub/child components. A *composite component* is a component composed of one or more sub/atomic/child components by exploiting the component composition concepts of software architectures [6, 18].

3-3 EVOLUTION TOOL[3]

The tool supports pattern-based and automated evolution of the architecture model. It is composed of an *Interface* and a *Core*. The *Interface* allows the designer/architect to import the architecture model and select the change patterns from the catalogue, as well as export the evolved model. The core provides the necessary functionality by applying constraints and operators and using a pattern to evolve the architecture model. The tool first interprets architecture model (Figure 4 A)) through its architectural descriptions (Figure 4 B)), then applies changes to descriptions and finally presents the evolved model to the architect. The tool is developed as a plugin for possible future extensions and utilising the resources of an integrated development environment (IDE). In contrast to a standalone system, the plugin can exploit the features and other plugins from IDE for stability, reusability of IDE resources, public availability and possible future extensions [8].

The relationships of elements of Evolution Tool with Architecture Model and Change Pattern are specified as: *Evolution Tool* **Import** *Architecture Model, Change Pattern*, as importing the necessary elements before evolution as well as *Evolution Tool* **Export** *Architecture Model*, exporting the new architecture model after evolution. We provide details of tool-support for pattern-driven and automated evolution in Section 4.

Table 2 Application of Child Creation Pattern

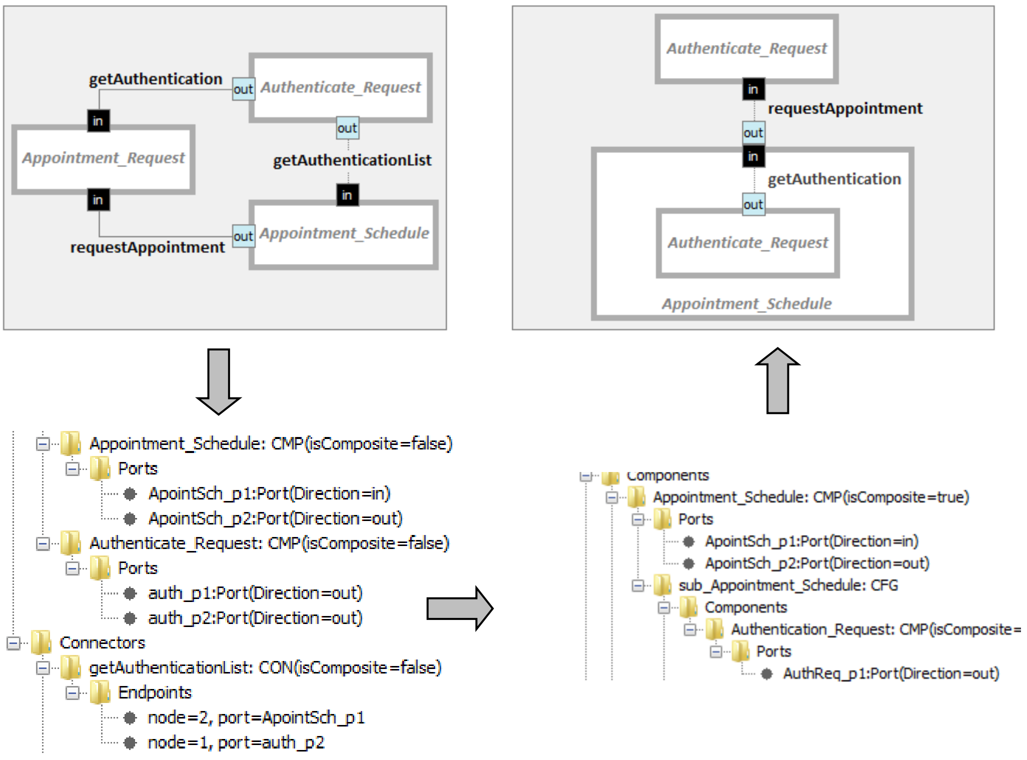
Child Creation Pattern
Pattern Intent: create a child component (X1) inside an atomic component (C1), such that C1 ⊳ X1 as C1 contains (⊳) X1 and C1 is a composite component now. **Evolution Scenario:** In Figure 3 P2P-AS architecture, client has to first Authenticate itself in order to request the Appointment. Moreover, the Appointment components must maintain a list of authenticated clients from Authenticate component. As the number of requesting client grows system performance decreases. The problem reflects: ***how to evolve the architecture of the appointment system to maintain its performance under increasing client requests?*** **Solution:** A possible solution to minimize excessive component interaction is with unification of the authentication and appointment functionalities (eliminating an explicit authentication and requesting for authenticated client's messages). The **Child Creation** pattern from the catalogue (cf. Table 1) can provide a solution (abstracting changes and constraints) with a high-level overview. The pattern creates the **Authenticate_Request** component as a child of **Appointment_Schedule** component. Such a change minimizes redundant interactions and also facilitates future modification easily.
Pattern Overview: The example scenario is provided in Figure 4 as an instance of the ChildCreation pattern from Table 1. - Figure 4 A) represents the source architecture model of P2P-AS (on left – preconditions) and its evolution (arrow representation) as a consequence of ChildCreation Pattern (on right – post-

[3] Please note the term Tool in this paper refers to a prototype as a preliminary/proof-of-concept version for our ongoing work on automated evolution of software architectures.

condition). The pattern in Figure 4 A) represents an abstraction with a generic overview of the architecture model before and after evolution.

- The refinement of the changes with addition or removal of individual component and ports in Figure 4 B) are reflected as changes in architecture description. In Figure 4 B) the **Appointment_Schedule** component is modified from an atomic to a composite component with fine-grained representation of change at the component, port, connector and endpoint level to ensure architectural composition is preserved.

A. Evolution of Architecture Model (abstract view)



```
Appointment_Schedule: CMP(isComposite=false)
    Ports
        ApointSch_p1:Port(Direction=in)
        ApointSch_p2:Port(Direction=out)
Authenticate_Request: CMP(isComposite=false)
    Ports
        auth_p1:Port(Direction=out)
        auth_p2:Port(Direction=out)
Connectors
    getAuthenticationList: CON(isComposite=false)
        Endpoints
            node=2, port=ApointSch_p1
            node=1, port=auth_p2
```

```
Components
    Appointment_Schedule: CMP(isComposite=true)
        Ports
            ApointSch_p1:Port(Direction=in)
            ApointSch_p2:Port(Direction=out)
        sub_Appointment_Schedule: CFG
            Components
                Authentication_Request: CMP(isComposite=
                    Ports
                        AuthReq_p1:Port(Direction=out)
```

B. Changes in Architecture Description (refinement)

Figure 4 Evolution Scenario – Evolution of Child Creation Pattern To Evolve P2P-AS.

4- PATTERN DRIVEN AND TOOL SUPPORTED EVOLUTION OF SOFTWARE ARCHITECTURES

In this research we hypothesised that by unifying change patterns and tool support, architectural changes can be (i) reused and (ii) automated for evolution of software architectures. In this section we aim to evaluate the hypothesis and demonstrate pattern-based and tool supported change management to evolve P2P-AS architecture. The evolution of the P2P-AS architecture from

Figure 4 to a client server model in Figure 5 enables *multiple clients, simultaneously requesting appointment schedules from server*. We demonstrate the usability of the tool to automate pattern-based changes and enabling the designer/architect to intervene, if and when required. We further evaluate the relative measure of reusability for *change primitive vs changes pattern* in Section 5. An overview of the pattern-based and tool-supported evolution process is presented in Figure 5 that comprises of three primary activities including (i) importing and exporting architecture descriptions, (ii) specification of architecture changes, and (iii) application of change pattern. In developing a software-intensive system (i) process illustrates *what needs to be done*, and (ii) activities in the process demonstrate *how it is done* [23]. All activities of the evolution process are detailed in Figure 5.

4-1 ACTIVITY I – IMPORTING/EXPORTING ARCHITECTURE DESCRIPTIONS

It allows us to import the architectural descriptions, known as source architecture model that can be evolved. Once the changes are applied, the evolved architecture model known as the target architecture can be exported from the tool to share its descriptions. For example, in Figure 5 A) the descriptions for source architecture model (from Figure 4) are imported that consist of two atomic components Appointment_Request and Authenticate_Request, whereas the Appointment_Schedule is a composite component composed of Authenticate_Request. The Appointment_Request and Appointment_Schedule are interconnected using requestAppointment connector. In Figure 5 A), the architect views source architecture model as the *graphical representation* on right with components and connectors view on the interface of tool, while source architecture description as *xml representation on left are* manipulated in the core of the tool. The architecture model view also allows the architect to add or remove any architectural components and connectors by using the ***Edit Architecture Model*** option.

4-2 ACTIVITY II – SPECIFYING ARCHITECTURAL CHANGES

It allows the architect to specify the (i) intended change(s) and (ii) any constraints on the source model – also referred to as user intervention in the evolution process.

- *Specifying Change Intent:* currently, the intent of change is specified by means of some predefined change rules (e.g. integration, composition, decomposition) of architectural components. For example, in Figure 5 B) the architect is provided with a user interface to specify the '*integration of a mediator component between the* Appointment_Request *and* Appointment_Schedule'. After selecting the intent of change, the architect specifies the impact of changes in terms of selecting the components to be evolved as the addition of an atomic component Appointment_Server. They can specify if they want to add an atomic or composite component by checking/unchecking the ***Composite*** option.

In the architecture model, the sub-architecture that is candidate for change is selected by the designer/architect. The sub-architecture is selected based on a declarative specification of the architectural changes by means of predefined change discussed above. For example, the integration of new architectural component named **Appointment_Server** between two existing components **Appointment_Request** and **Appointment_Schedule** is specified by the user as:

Integrate(Appointment_Server, (Appointment_Request, Appointment_Schedule) $\sqsubseteq$ CMP).

The rule specifies that existing sub-architecture in terms of two architectural elements as components **Appointment_Request**, **Appointment_Schedule**, whereas **Appointment_Server** is a new component to be integrated in the sub-architecture.

- *Specifying Change Constraints*: after specifying the changes on architecture model, the architect can also specify the constraints as the post-conditions of the evolution, while the preconditions are automatically computed based on the source architecture descriptions. For example, the architect specifies post-condition as the integration of a new component **Appointment_Server** along with two connectors **requstAppointment** interconnecting **Appointment_Server** and **Appointment_Request** where **requestSchedule** binds **Appointment_Server** and **Appointment_Schedule** components. Once the changes are specified the architect can choose to *Select Patterns* to retrieve the most appropriate patterns from the catalogue (cf. Table 1). The patterns are selected automatically based on the preconditions and post-conditions. These pre/post-conditions specified by the architect are expressed as XML constraints and are matched with the pre/post-conditions of a pattern (cf. Figure 4).

It is vital to mention if a pattern is not found such as pre/post-conditions specified by the architect do not match to those of pattern, the tool allows the architect to manually specify the changes (***Edit Architecture Model*** option as a case of change primitives [5]). Matching of the preconditions is a mandatory condition for the selection of the sub-architecture for change implementation.

4-3 ACTIVITY III – APPLYING CHANGE PATTERNS

Once changes are specified (Activity II) on architecture model (Activity I); based on change specification and constraints, the appropriate pattern is displayed with its impact on the architecture. For example, in Figure 5 C) the **Component Mediation** is selected that enables the integration of **Appointmrnt_Server** to mediate between **Appointment_Request** and **Appointment_Schedule** components. The view in Figure 5 C) helps an architect to visualise the impact of a pattern on the architecture to decide if it results in desired impact of change on the model or not. The user can enable pattern refinements by adding the component ports and refinement can be seen as an auxiliary step to pattern application - adding port level details to architectural components (cf. Figure 4). Finally, when the architect aims to apply pattern he/she can click on ***Apply Pattern*** to execute changes, otherwise pattern application can be cancelled and no change will be applied to the source architecture model.

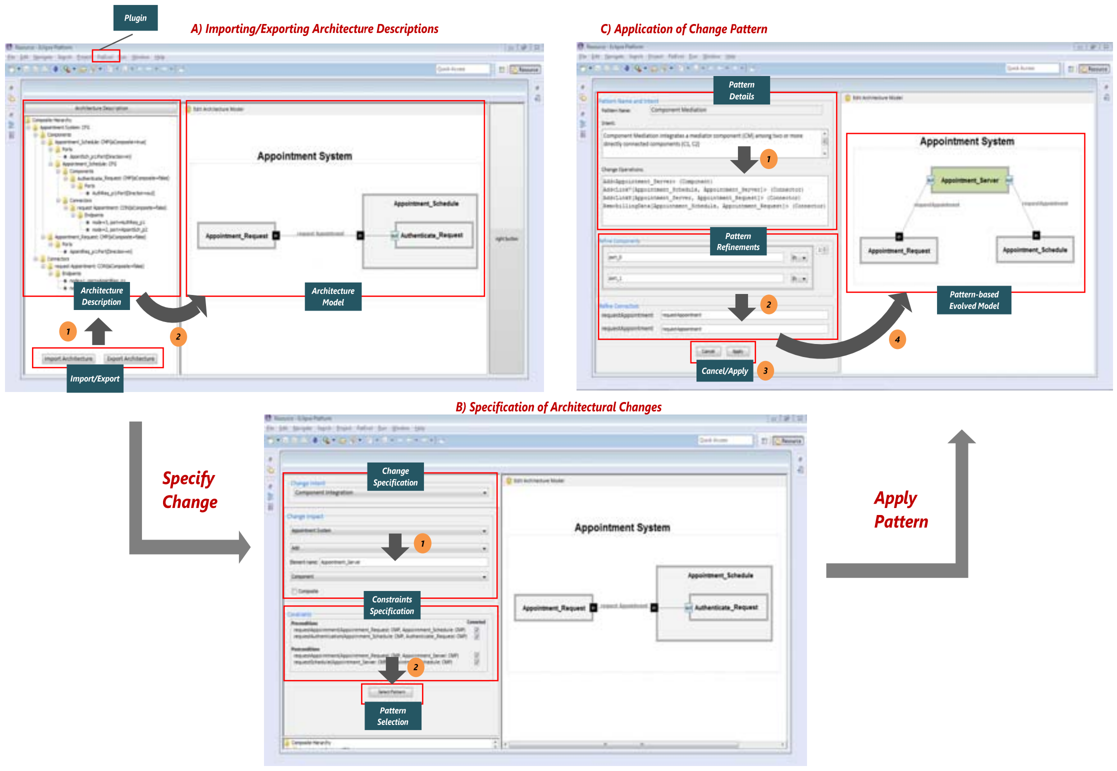


Figure 5. Overview of the Pattern-based and Tool Supported Evolution.

5- EVALUATIONS, LESSONS LEARNT AND VALIDITY THREATS

Based on the details in Section 4 that focused on demonstrating tool support, we now report results of evaluation for pattern-based reuse of the architectural changes.

5-1 EVALUATING PATTERN-BASED CHANGE REUSE

We utilise the ISO/IEC 25010:2011 quality model to evaluate the reusability of architectural changes [3]. The work on pattern-based evolution is a continuous effort, therefore; results presented here and elsewhere [11, 25] represent an incremental effort for a step-wise evaluation of the overall solution – i.e; pattern discovery to complement pattern application [7]. Reusability as a sub-characteristic of maintainability represents the quality of solution to enable the reuse of existing resources in terms of knowledge/expertise to address the recurring evolution tasks [3]. To evaluate reusability of architectural evolution, we use comparison of primitive vs proposed pattern-based changes with the following metric adopted from [24] and extended for our needs:

Total Change Operations (TCO) - to quantify the required efforts for architectural change implementation, we derived and used the TCO metric as "*the total number of architecture change operations required to resolve an architecture evolution scenario.* TCO can be viewed as an inspiration from the Line of Codes (LOC) metric that estimates the total lines of code required to achieve the desired functionality or represent the size or magnitude of source code. In contrast, the TCO measures the total number of change operations (i.e; operational size of evolution). To measure a relative reusability *(Primitive: $Primitive_{TCO}$ vs Pattern: $Pattern_{TCO}$),* we formulate the ratio as follows:

$$(1 - Pattern_{TCO} / Primitive_{TCO}) \text{ X } 100$$

Figure 6 provides a comparison overview of the relative number of change operations for implementing a specific change as resolving an evolution scenario. For example, to support the change intent that supports integration of components the *Component Mediation* pattern (cf. Figure 5) requires three operators in terms of the preconditions, pattern application and the post-conditions also reducing the effort of pattern selection performed by the tool. In contrast, to implement similar change the change primitives require at-least a total of five operations to add the required component, their interfaces and connectors. In addition, with change primitives the designer/architect must be well aware of available patterns to rely on his knowledge for the selection of the most appropriate patterns that is a significant challenge for novice architects [15]. In this scenario, the *ratio of reuse is given as R = 1 – (3/5) x 100 calculated as 40%*. To generalise the findings, a summary of the comparison for patterns vs primitives (using TCO) is provided in Figure 6, we measure a total of 34% approx. reuse by applying patterns on the P2P-AS evolution case study. Based on the summary of results in Figure 6, we provide an overview of the comparative analysis for TCO for primitive and pattern-based changes.

Pattern-based changes take approximately 30% – 35% of change operations compared to primitive changes. However, pattern-based change does not support a fine granular change representation. An increase in the percentage reuse is (i) directly proportional to an increase in the total number of changes, and (ii) inversely proportional to change implementation efforts as primitive vs pattern based change.

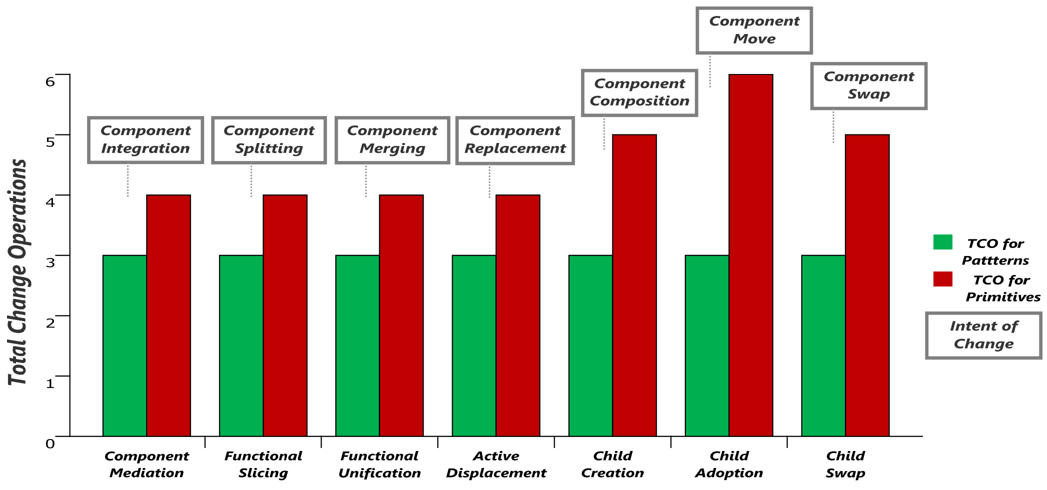
Consideration for Total Time Taken: Another evaluation is based on the time efficiency as the time taken for change implementation that is considered as part of future evaluations. We have primarily focused on exploiting patterns as reusable solutions to support design-time evolution; however, to support runtime evolution patterns must be validated to support cost/time-efficient implementation of changes as part of our on-going and future research.

5-2 LESSONS LEARNT

We discuss a few lessons learnt based on architectural evolution and the lessons can also indicate future refinements of the solution. We generalise the presentation such that any architecture models with component-connector view that needs to be evolved can also benefit from our observations and lessons.

Lesson I - Granularity Conflicts Reusability: In primitive vs pattern-based modifications, there is a trade-off between the granularity and reusability. Granularity of architectural modification refers to the completeness of modifications such as adding configurations with components that contain ports, etc. Reusability of architectural changes refers to reuse of generic modification operations. In contrast to primitive architectural changes, pattern-based changes support reuse with enhanced efficiency of modification process resulting in an average 33% reuse. However, pattern-based changes do not support a fine-granular change implementation. A possible solution is to introduce the pattern refinements that extend the higher-level architectural components and connectors as composite elements with interfaces and endpoints respectively as atomic elements [17].

Lesson II – Consistency vs Reusability of Changes: The role of pattern catalogue is central in promoting patterns for reuse and consistency of architectural modification.



	Component Mediation	Functional Slicing	Functional Unification	Active Displacement	Child Creation	Child Adoption	Child Swap
TCO for Pattterns	3	3	3	3	3	3	3
TCO for Primitives	4	4	4	4	5	6	5

Figure 6. A comparison of Total Change Operations - Primitives vs Patterns.

During architectural modifications, the structural consistency of architecture model must be maintained by preserving the architectural hierarchy (part-whole relation) as modeled Figure 2. For example, a component that is not connected to another due to a missing port, endpoint or connector is called an orphaned component that violates architectural quality and structure. Orphaned component results in counter-productive and negative impacts of patterns that may result in violating architectural composition. The model in Figure 2 is the basic mechanism to confirm a valid pattern and architecture model. In addition, the invariants help in preserving the required architectural structure and properties in an attempt to minimise the violation of architectural composition.

Lesson III – The Needs for Usability Evaluations from Practitioners: Usability as a system quality must support the attributes like learnability, operability, etc. by its users [3]. In Section 4, we only manage to *demonstrate* the tool's usability, while *evaluating* usability requires an empirical evaluation by engaging multiple designers/architects to utilise the tool and report their experiences about its usefulness. Given the scope of research we envision usability evaluation based on experts' feedback as essential criteria to further strengthen the solution and also as one of the dimensions of future research.

5-3 VALIDITY THREATS

In this section we discuss the threats to the validity of this research that can become possible limitations and provide an indication of future work that can possibly minimise these threats.

Threat I – Limited Data for Architecture Evolution: A limited number of existing change patterns and a small case study to evaluate pattern applicability represents a validity threat. The prospects of evaluating the effectiveness of our solution for industrial case studies requires more data and validation to comment on the benefits of solution in the context of evolution for industrial software. Specifically, in an industrial scale software architecture, the evolution usually takes place over long periods that span months, years and often decades [1, 13]. In the general context, the research aims to provide a foundation with a framework that integrates architecture change mining for a continuous acquisition of knowledge as patterns that support reusability and efficiency of the change execution process.

Threat II – Accuracy of Pattern Selection: During pattern-based architecture evolution, accuracy of the pattern selection refers to solution's ability to select the most appropriate change patterns from a pattern collection. The possible threat to a more rigorous validity of pattern selection is the limited number of patterns in the pattern collection (cf. Table 1). This threat has a direct impact on selecting the most appropriate patterns from pattern collection. Currently, we have a total of 7 change patterns that represent a relatively limited number of patterns. As the number of change patterns in the pattern catalogue grows it may have an impact on the precision of pattern selection.

6- RELATED RESEARCH

A recent study on classification of architectural evolution research in [12] has highlighted [18] as one of the earliest studies, published in 1995 to support software evolution and refinement through its architecture. Since then, architectural evolution research has progressed and matured over two decades with a collective impact of research highlighted in recent studies [4, 12]. We specifically position our contributions in the context of reusable (Section 6.1) and automated (Section 6-2) evolution that represent two of the vital challenges both from academic and from industrial perspective [12, 14]. There is no existing solution on unifying patterns and tool support for architectural evolution.

6-1 REUSABLE CHANGE MANAGEMENT FOR SOFTWARE ARCHITECTURES

A) ACADEMIC RESEARCH

Recently, the concepts of change patterns [19, 20, 26] and evolution styles [6] have emerged as solutions - focused on applying reusable knowledge and expertise - to address the recurring problems of architectural evolution. Change patterns are further classified as patterns that support the (i) design-time evolution of requirements and corresponding architecture model and (ii) adaptation patterns that support runtime evolution of architectures [19, 20]. The notion of change patterns is inspired by design patterns, however; instead of design concerns, change patterns specifically address architectural maintenance and evolution as *corrective*, *adaptive* and *perfective* changes as per ISO/IEC change taxonomy [20]. In contrast, evolution styles propose evolution planning to derive reusable evolution paths [6]. An evolution path represents a reusable strategy to plan architectural evolution based on the cost, time, and efforts of evolution. Evolution styles are based on the classical concept of architecture styles.

B) INDUSTRIAL STUDIES AND SOLUTIONS

In an industrial scale evolution, that is driven-by various types of constraints including time and economic aspects; there is a growing interest on exploiting potentially reusable changes that replace ad-hoc and once-off changes. One of the recent examples is [14] that support the reuse of adaptation policies to support run-time evolution of an industrial system called Data Acquisition and Control Service (DCAS). DCAS system monitors and manages highly populated networks of devices in renewable energy production plants. The research has demonstrated the reuse of recurring adaptation strategies that minimised about 40% of the efforts for architecture evolution compared to an ad hoc and once-off change implementation. In [13] an empirical study of change requests has been conducted based on four different releases of a large telecom system. The findings of the study highlights that change reuse have resulted in an (i) increased maintainability, testability and overall quality with (ii) decreased efforts and time for future implementation of changes.

6-2 AUTOMATED EVOLUTION OF SOFTWARE ARCHITECTURE

In comparison to reuse, the automation of architecture change management has achieved much less attention with only notable studies [21, 22]. Patterns facilitate reuse, however; a lack of automation burdens an architect with manual – exhaustive, error-prone and time-consuming – efforts to plan and execute evolution [19]. Automation of such tasks seems ideal, however; it entails certain challenges like complexity, type of architecture and evolution that must be supported by a tool [8]. Moreover, as identified in [21] architecting and evolution are more than just enabling the automation; instead an appropriate tool support must complement the user/architect's intervention to guide the change management process.

Enabling evolution and its automation require iterative development of solution. Specifically, evolution reuse relies on strong theoretical foundations of its usefulness before developing any tools to support automation. A recent study is considered as the first attempt to establish evolution styles as artifacts of reuse [6]. Once the foundation and usefulness have been identified that provides the requirements and needs to develop a tool for an automated specification of styles [22]. The research on evolution of automated production systems highlights the challenges and futuristic research dimensions of continuous change management of industrial scale software systems [27]. The study identifies that, in order to support a continuous maintenance and evolution of complex and real world software, automation can be influential for time and cost-effective change management.

Comparison Summary: Existing vs Proposed Solution - In conclusion, the proposed solution generally lies at the intersection of software reuse and automated software engineering. First, the proposed solution is relevant to research that focus on avoiding ad-hoc changes with reusable and best practices for software evolution [6, 14]. In contrast to these studies, the proposed solution promotes a two-step process for reusable evolution with (i) change pattern discovery, (ii) exploiting discovered patterns from [15, 25] to enable reuse of architectural changes. However, in contrast to [14]; our solution is focused on design-time rather than runtime evolution that can be considered as a concern for future research. Moreover, we aim to complement the existing research with reuse and automation of architecture evolution [21, 22]. Pattern languages and pattern catalogues for architecture evolution can benefit from tool support [15]. We demonstrated that patterns alongside tool support provide an effective mechanism for reusable and automated change support, while incorporating the architect's intervention and feedback is vital to guide the evolution process. We have also followed the same two step approach as

(i) establishing and discovering architecture change patterns (previous work [11, 25], cf. Section 3), followed by (ii) work in this paper that supports application of discovered patterns to automate evolution (cf. Section 4).

7- CONCLUSIONS

In this paper, we propose to support changes in software architecture by means of change patterns exploited with an appropriate tool support. The existing research have exploited patterns as artifacts of reuse or tool support to enable automation, however; there is no research to unify pattern-driven and tool-supported architectural evolution. We have focused on the unification of the empirically discovered change patterns and systematically developed tool support to enable reusable and automated evolution of software architectures.

The preliminary results of evaluation suggest that pattern-based changes enable reuse but lack a fine-granular change execution. The granularity of change representation refers to completeness of change execution on architectural elements. Specifically, architecture change patterns abstract the primitive changes that add/remove/modify components and connectors into reusable pattern-based changes to compose/merge/replace components and connectors. Tool-support facilitates automation of architecture change implementation, however; user intervention is critical to guide architectural evolution. We have also identified some validity threats like availability of data and accuracy of pattern selection. Our future work primarily focuses on acquisition of data and involvement of practitioners for further validations and more objective interpretation of the results.

REFERENCES

[1] T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer, 2008.

[2] M. M. Lehman and J. F. Ramil, “Software Evolution: Background, Theory, Practice,” Information Processing Letters, vol. 88, no. 1, pp. 33–44, 2003.

[3] ISO, “ISO/IEC 25010:2011 Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models,” 2011. [Online:] http://webstore.iec.ch/preview/info_isoiec25010%7Bed1.0%7Den.pdf

[4] H. P. Breivold, I. Crnkovic, and M. Larsson, “A Systematic Review of Software Architecture Evolution Research,” Information and Software Technology, vol. 54, no. 1, pp. 16–40, 2012.

[5] B. J. Williams and J. C. Carver, “Characterizing Software Architecture Changes: A Systematic Review,” Information and Software Technology, vol. 52, no. 1, pp. 31–51, 2010.

[6] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, “Evolution Styles: Foundations and Tool Support for Software Architecture Evolution,” in Joint Working IEEE/IFIP Conference on Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ ECSA. IEEE, 2009, pp. 131–140.

[7] A. Ahmad, P. Jamshidi, and C. Pahl, “Classification and Comparison of Architecture Evolution-Reuse Knowledge - A Systematic Review,” in Journal of Software: Evolution and Process. DOI: 10.1002/smr.1643. Wiley, 2014.

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

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

[10] O. Zimmerman, J. Koehler, F. Leymann, R. Polley, and N. Schuster, N. 2009. Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. In Journal of Systems and Software. Vol 82, pp: 1249 – 1267, 2009.

[11] A. Ahmad, P. Jamshidi, and C. Pahl, “Graph-based Pattern Identification from Architecture Change Logs,” in In Tenth International Workshop on System/Software Architecture. Springer, 2012, pp. 200–213.

[12] P. Jamshidi, M. Ghafari, A. Aakash, and C. Pahl, “A Framework for Classifying and Comparing Architecture-centric Software Evolution Research,” in 17th European Conference on Software Maintenance and Reengineering (CSMR’13). IEEE, 2013, pp. 305–314.

[13] P. Mohagheghi and R. Conradi, “An Empirical Study of Software Change: Origin, Acceptance rate, and Functionality vs. Quality Attributes,” in nternational Symposium on Empirical Software Engineering, ISESE’04. IEEE, 2004, pp. 7–16.

[14] J. C´amara, P. Correia, R. De Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura, “Evolving an Adaptive Industrial Software System to Use Architecture-based Self-adaptation,” in 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. IEEE, 2013, pp. 13–22

[15] A. Ahmad, P. Jamshidi, C. Pahl, F. Khaliq. A Pattern Language for Evolution Reuse in Component-based Software Architectures. ECASST Special Issue on Patterns Promotion and Anti-patterns Preventions, vol 59, pp: 1

– 32, 2013.

[16] N. Medvidovic, and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Transactions on Software Engineering, vol 26, issue 1, pp: 70-93, 2000.

[17] Harrison, N. B., Avgeriou, P., Zdun, U.: Using Patterns to Capture Architectural Decisions. In IEEE Software 24(4): 38-45, 2007.

[18] M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct Architecture Refinement. In IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 356 - 372, 1995.

[19] K. Yskout, R. Scandariato, and W. Joosen, "Change patterns: Co-evolving Requirements and Architecture. In Journal of Software and Systems Modeling, vol 13, no 2, pp: 625-648, 2014.

[20] H. Gomaa, K. Hashimoto, M. Kim, M, S. Malek, D. A. Menascé, Software Adaptation Patterns for Service-oriented Architectures. In 2010 ACM Symposium on Applied Computing, ACM, 2010.

[21] L. Grunske. Automated Software Architecture Evolution with Hypergraph Transformation. In Proceedings of the 7th International IASTED on Conference Software Engineering and Application, 2003.

[22] J.M. Barnes, A. Pandey, and D. Garlan. Automated Planning for Software Architecture Evolution. In 28th IEEE/ACM International Conference on Automated Software Engineering, 2013.

[23] M. E Fayad. Software Development Process: A Necessary Evil. In Communications of the ACM, vol. 40, no. 9, pages 101–103, 1997.

[24] S. Tragatschnig, H. Tran, and U. Zdun. Change patterns for supporting the evolution of event-based systems. In 21st International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS 2013), Springer, 2013

[25] A. Ahmad, P. Jamshidi, C. Pahl. Graph-based Discovery of Architecture Change Patterns from Logs. Technical Report, School of Computing, Dublin City University, 2012.
[Online:]
http://media.wix.com/ugd/396772_b1edc14eec2a4567b0f4e38a4e364653.pdf

[26] I. Côté, M. Heisel, and I. Wentzlaff. 2007. Pattern-based Evolution of Software Architectures. Lecture Notes on Computer Science, vol. 4758, pp.29-43, 2007.

[27] B. Vogel-Heuser, A. Fay, I. Schäfer and M. Tichy. Evolution of Software in Automated Production Systems — Challenges and Research Directions. In Journal of Systems and Software , vol 110, pp: 54 - 84, 2015.