A Pattern Language for Evolution Reuse in Component-Based Software Architectures

Aakash Ahmad Abbasi

BS in Software Engineering (IIUI) 2008

A Dissertation submitted in fulfilment
of the requirements for the award of

Doctor of Philosophy (Ph.D.)

to the

DUBLIN CITY UNIVERSITY
FACULTY OF ENGINEERING AND COMPUTING
SCHOOL OF COMPUTING

Supervisor: Dr. Claus Pahl

August, 2015

Declaration

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Doctor of Philosophy is entirely my own work, that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge breach any law of copyright, and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ——————————

Aakash Ahmad Abbasi

Student ID : 58111221

Date: ——————————-

A Pattern Language for Evolution Reuse in Component-Based Software Architectures

Abstract

Context: Modern software systems are prone to a continuous evolution under frequently varying requirements and changes in operational environments. Architecture-Centric Software Evolution (ACSE) enables changes in a system's structure and behaviour while maintaining a global view of the software to address evolution-centric trade-offs. Lehman's law of continuing change demands for long-living and continuously evolving architectures to prolong the productive life and economic value of software. Also some industrial research shows that evolution reuse can save approximately **40%** effort of change implementation in ACSE process. However, a systematic review of existing research suggests a lack of solution(s) to support a continuous integration of reuse knowledge in ACSE process to promote *evolution-off-the-shelf* in software architectures.

Objectives: We aim to unify the concepts of software repository mining and software evolution to discover evolution-reuse knowledge that can be shared and reused to guide ACSE.

Method: We exploit repository mining techniques (also *architecture change mining*) that investigates architecture change logs to discover change operationalisation and patterns. We apply software evolution concepts (also *architecture change execution*) to support pattern-driven reuse in ACSE. Architecture change patterns support composition and application of a pattern language that exploits patterns and their relations to express evolution-reuse knowledge. Pattern language composition is enabled with a continuous discovery of patterns from architecture change logs and formalising relations among discovered patterns. Pattern language application is supported with an incremental selection and application of patterns to achieve reuse in ACSE. The novelty of the research lies with a framework PatEvol that supports a round-trip approach for a continuous acquisition (mining) and application (execution) of reuse knowledge to enable ACSE. Prototype support enables customisation and (semi-) automation for the evolution process.

Results: We evaluated the results based on the ISO/IEC 9126 - 1 quality model and a case study based validation of the architecture change mining and change execution processes. We observe consistency and reusability of change support with pattern-driven architecture evolution. Change patterns support efficiency for architecture evolution process but lack a fine-granular change implementation. A critical challenge lies with the selection of appropriate patterns to form a pattern language during evolution.

Conclusions: The pattern language itself continuously evolves with an incremental discovery of new patterns from change logs over time. A systematic identification and resolution of change anti-patterns define the scope for future research.

Keywords: *Software Evolution, Software Architecture, Architecture-Centric Software Evolution, Software Repository Mining, Pattern Discovery, Change Patterns, Pattern Language*

Acknowledgments

Alhamdulillah,

I am extremely thankful to my Ph.D. supervisor, *Dr. Claus Pahl* for his continuous patience, encouragement, guidance and support throughout my research. Claus thank you very much for all your support, helping me to think more independently and encouraging to freely share and develop ideas beyond the thesis topic.

Thank you to people at Lero - the Irish Software Engineering Research Centre for all the support throughout and for fully funding my Ph.D. research.

I feel fortunate enough to be a part of the Software and System Engineering Research Group at DCU. *Veronica*, *Wang*, *Kosala* I thank you very much for being open and accommodating towards me and enlightening me with different cultures and interesting discussions. *Javed* and *Yalemisew* thank you for your inspiring ideas and technical feedback during our discussions in CNGL boardroom. Thank you *Imran* and *Khalid* for your support and company.

My early days in Ireland - *Paul* and *Oísin*, I am sincerely thankful to you for introducing me to the Irish culture and helping me to settle down. *Ray* thank you very much for all your support and encouragement during my Ph.D.

Pooyan and *Hourieh*, I could never thank you enough for looking after me with an immense warmth, and being my family while away from home. *John* and *Huanhuan* it was always a pleasure to visit you.

Claus and Pooyan, you made me believe that doing research is all about satisfaction and passion, thank you for sharing mine!

Fatimah, thank you for your love and support. My *dad*, *brother* and *Hamza* at home.

To **Ammi** - *no words to express, you are and you will be the biggest inspiration!*

List of Abbreviations

Abbreviations	Description
ACSE	Architecture-centric Software Evolution
AERK	Architecture Evolution Reuse Knowledge
AK	Architecture Knowledge
ACL	Architecture Change Log
CBSA	Component Based Software Architecture
GOF	Gang-of-Four
GPride	Graph-based Pattern Identification
PatEvol	Pattern-driven Architecture Evolution
QOC	Question Option Criteria
UML	Unified Modeling Language
MOF	Meta Object Facility
AG	Attributed Graph
ATG	Attributed Typed Graph
GML/GraphML	Graph Modeling Language
DCAS	Data Acquisition and Control Service
COTS	Commercial-Off-The-Shelf
CBSE	Component Based Software Engineering
SOSE	Service Oriented Software Engineering
ADL	Architecture Description Language
SPO	Single Push Out
DPO	Double Push Out
EBPP	Electronic Business Presentment and Payment
CS-AS	Client Server Appointment System
SLR	Systematic Literature Review
SEI	Software Engineering Institute
SOA	Service Oriented Architecture
SOA-MF	SOA Migration Framework
ADM	Architecture Driven Modernisation
STAC	Software Tuning Panels for Autonomic Control
MAPE-K	Monitor, Analyse, Plan, Execute Knowledge
IBM	International Business Machine
RQ	Research Question
HCI	Human Computer Interaction
ARCH	Architecture Model
OPR	Operations
CNS	Constraints
PAT	Pattern
COL	Collection
PRE	Preconditions
POST	Postconditions
INV	Invariants
CD	Change Data
AD	Auxiliary Data
CFG	Configuration
CMP	Component
CON	Connector
POR	Port
EPT	Endpoint
LenEqu	Length Equivalence
OrdEquv	Order Equivalence
TypEquv	Type Equivalence
BFS	Breadth First Search
ALMA	Architecture Level Modifiability Analysis
ISO	International Organisation for Standardisation
TCO	Total Change Operations
AMS	Auction Management System
SLA	Service Level Agreement
QoS	Quality of Service
ECA	Event Condition Actions

Glossary of Definitions

- **Architecture Evolution Reuse Knowledge** is defined as a collection and integrated representation (problem-solution mapping) of empirically discovered change implementation expertise that can be shared and reused as a solution to frequent evolution problems.
- **Reuse Knowledge Acquisition** is defined as the process to systematically discover reuse knowledge (change operations and patterns) that can be shared for future reuse.
- **Reuse Knowledge Application** is defined as the process to systematically apply reuse knowledge (change operations and patterns) to support evolution of architectures.
- **Architecture Change Log** is defined as a repository infrastructure with fine-grained representation of architecture evolution by capturing the intent, scope and operationalisation of individual architectural changes.
- **Attributed Graph** contains a set of nodes and edges. An attributed graph is defined as a type of graph in which we can associate a number of attributes to the nodes and edges of the graph.
- **Change Log Graph** is defined as a graph that represents the change log data as a graph. In change log graph architectural changes are represented as graph nodes, while the sequencing among change operations is maintained with graph edges.
- **Atomic Change Operation** is defined as an operation that supports a single change on an individual architecture element. It represents the most fundamental unit of architectural change to enable evolution.
- **Composite Change Operation** is defined as a collection of atomic change operations to support a collection of architectural changes. Composite changes abstract the details from individual atomic changes.
- **Architecture Change Primitives** are defined as the types of architectural changes that support the addition, removal and modification of components and connectors in architectural configurations.
- **Architecture Change Pattern** is defined as a collection of generic and reusable change operationalisation that can be discovered as recurrent, specified once and instantiated multiple times to support reuse in architecture evolution.
- **Change Pattern Language** is defined as a collection of interconnected patterns that supports an incremental application of patterns to support reuse in architecture evolution.
- **Pattern Language Vocabulary** is defined as the collection of discovered patterns and their possible variants.
- **Pattern Language Grammar** is defined as the structure of pattern language that support possible interconnections among patterns in the language.
- **Pattern Language Sequencing** is defined as an ordered sequence in which pattern can be selected and applied in a pattern language.

Selective Publications

(Full list at: `http://ahmadaakash.wix.com/aakash#!publications/c1p03`)

Journal Articles

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl. *Classification and Comparison of Architecture Evolution-Reuse Knowledge - A Systematic Review*. Journal of Software: Evolution and Process, vol 26, issue 7, pp: 654 - 691, 2014.

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl, Fawad 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: 2 - 32, 2013. (invited paper)

Conference and Workshop Papers

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl, Fawad Khaliq. *PatEvol - A Pattern Language for Evolution in Component-based Software Architectures*. In 1st International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP, co-located with CSMR), 2013.

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl. *A Framework for Acquisition and Application of Architecture Evolution Reuse Knowledge*. In ACM SIGSOFT Software Engineering Notes (SEN), vol. 38, no. 5, 2013.

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl. *Graph-based Implicit Knowledge Discovery from Architecture Change Logs*. In 7th Workshop on SHAring and Reusing architectural Knowledge (SHARK, co-located with WICSA/ECSA). ACM, 2012.

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl. *Graph-based Pattern Identification from Architecture Change Logs*. In 10th International Workshop on System/Software Architectures (IWSSA, co-located with CAiSE). Lecture Notes in Computer Science, 2012.

- **Aakash Ahmad**, Pooyan Jamshidi, Claus Pahl. *Pattern-driven Reuse in Architecture-centric Evolution for Service Software*. In 7th International Conference on Software Paradigm Trends (ICSOFT). SCITEPRESS Digital Library, 2012.

- **Aakash Ahmad**, Claus Pahl. *Pat-Evol: Pattern-Driven Reuse in Architecture-Based Evolution for Service Software*. In ERCIM News 88, Special Issue on Software Evolution, 2012.

- **Aakash Ahmad**, Claus Pahl. *Customisable Transformation-driven Evolution for Service Architectures*. In 15th European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society, 2011.

- **Aakash Ahmad**, Claus Pahl. *Pattern-based Customisable Transformations for Style-based Service Architecture Evolution*. In 6th International Conference on Next Generation Web Services Practices (NWeSP). IEEE Computer Society, 2010.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Architecture-Centric Software Evolution

1.1.1 Software Evolution

Software evolution as per ACM/IEEE software engineering curricula is defined as: *'a sub-domain of software engineering that aims to investigate and support methods and techniques to adapt existing software to evolving requirements'* [Mens 2008, Lehman 2003]. As a consequence of frequently changing requirements, modern software systems are prone to a continuous evolution [Lehman 1996, Breivold 2012]. The primary causes of software evolution[1] could be categorised as changes in stakeholders' needs, business and technical requirements and operating environments [Mens 2008, Yskout 2012]. Such changing requirements trigger a continuous evolution in software structure and behaviour [Sadou 2005] that needs to be addressed while maintaining a global-view-of-system to resolve evolution-centric trade-offs [Breivold 2012, Garlan 2009]. However, the problem of software evolution is strengthened primarily due to: a) *recurring nature of change* [Lehman 1996, Garlan 2009] and b) *selection of an appropriate abstraction* [Williams 2010, Sadou 2005] to implement such change. The recurring nature of change requires a continuous accommodation of evolving requirements in existing software to prolong its productive life and economic value [Mens 2008].

Lehman's law of 'continuing change' [Lehman 1996] poses a direct challenge for research and practices that aim to support long-living and continuously evolving software [Garlan 2009, Le Goaer 2008], under frequently varying requirements [Yskout 2012]. The law states that "*... systems must be continually adapted or they become progressively less satisfactory*". In addition, software is composed of multiple layers of abstraction that includes its source code [Moghadam 2012], design and architecture [Medvidovic 1999] along with application specific configurations [Sadou 2005]. Therefore the selection of an appropriate abstraction is also critical to facilitate modelling, analysis and execution of software changes in a systematic, efficient and cost-effective manner. The challenge lies with supporting a continuous change, with change implementation at an appropriate abstraction-level to manage evolution during software life-cycle [Williams 2010].

1.1.2 Software Architecture

Software architecture as per the ISO/IEC/IEEE 42010 standard is defined as: *'fundamental concepts or properties of a (software) system in its environment embodied in its elements, relationships, and in the principles of its design and evolution'* [ISO-IEC-IEEE42010 2011, Perry 1992]. During the design, development, and evolution of software systems, the role of an architecture as a blue-print of software is central to map the changes in requirements [Yskout 2012] and their implementation in source code [Moghadam 2012]. Architecture abstracts the implementation specific details of a software by modelling (low-level) lines-of-code to (high-level) architectural components and their interconnections [Medvidovic 1999, Bengtsson 1999]. Architectural models proved successful in representing modules-of-code and their interconnections as high-level components and connec-

[1] Please note that in existing literature the terms *software evolution* and *software change* are virtually synonymous and often used interchangeably [Lehman 2003, Williams 2010, Buckley 2005]. However, in this thesis a technical distinction must be maintained among the two. Implementation of a collection of changes on existing software leads to its evolution.

tors that facilitate planning, modelling and executing software evolution at higher abstractions [Le Goaer 2008]. A systematic classification and comparison of architecture-centric software evolution research [Breivold 2012, Jamshidi 2013b] highlights the role of architecture models as system abstractions to facilitate *analysis*, *planning*, *modelling*, and *execution* of architectural changes.

Once, the decision is made about exploiting architectural abstractions to address software evolution, the implications of continuing change [Lehman 1996] demand frequent evolution to avoid architectural degradation [Williams 2010] (a.k.a. *architectural erosion* [Bengtsson 1999, Lassing 2003]). Research state-of-the-art [Zhang 2012] highlights the potential for solutions that enable acquisition and application of reuse knowledge to guide architectural evolution.

1.1.3 Evolution of Software Architecture

Architecture-centric Software Evolution (ACSE) is defined as: *'a technique to support evolution of a software system at its architectural level of abstractions'* [Garlan 2009, Breivold 2012]. Any attempts that aim to systematically address ACSE must rely on empirically discovered knowledge that can be shared and reused to manage evolution [Li 2012]. Our claim is based on a consolidated evidence of existing research gathered by conducting the systematic literature reviews [Jamshidi 2013b] on ACSE. We systematically analysed the collective impact, possible limitations and future potential of existing research on architecture evolution and architecture evolution-reuse knowledge [2]. Our findings based on a literature review highlight change patterns [Côté 2007, Yskout 2012] and evolution styles [3] [Garlan 2009, Tamzalit 2010] as the predominant solutions to facilitate architecture evolution reuse.

1.1.4 Reuse in Evolution of Software Architectures

In existing research, the reuse of change implementation is supported with Architecture Evolution-Reuse Knowledge (AERK). It could be argued that AERK is classified as a sub-domain of Architecture Knowledge (AK) [Zhang 2012]. However, a systematic mapping of architecture knowledge research [Li 2012] suggests minimal evidence of reuse-driven maintenance and evolution. This leads us to believe that in a general context of AK there is a need to explicitly focus on exploiting the potential, limitations and future trends for reuse knowledge and expertise to systemati-

[2] Please note that we use the terms *Architecture Evolution-Reuse Knowledge* and *Evolution-Reuse Knowledge* and *Reuse Knowledge* are used interchangeably - all referring to the same concept.

[3] In literature the terminologies and concepts of style and pattern are often used interchangeably referring to reusable artefacts for software design and development. In this thesis, we maintain a distinction between styles and patterns. A style represent a reusable vocabulary of architectural elements (component and connectors) and a set of constraints on them to express an architectural style. Patterns represents a generic, repeatable solution to recurring problems.

cally address recurring evolution in architectures. Moreover, a recent emergence of architecture evolution styles [Garlan 2009, Le Goaer 2008] and change patterns [Yskout 2012, Côté 2007] promoted the needs for research on the development of processes, patterns and frameworks that enable knowledge-driven evolution [Ulrich 2010] and adaptation [Ganek 2003] of architectures. Considering the general context of AK and specifically focusing on architecture evolution-reuse knowledge, we propose an integration of: a) *knowledge acquisition* and b) *knowledge application* processes to discover and apply evolution-reuse knowledge for architectural change implementation. In the context of ACSE, we define architecture evolution-reuse knowledge as: *domain specific problem-solution mapping that enables utilisation of reusable expertise to address frequent evolution problems*. The needs for reusable knowledge that supports architectural maintenance and evolution are acknowledge by the practitioners (software architects) [Clerc 2007, Mohagheghi 2004] and a recent industrial study [Cámara 2013] that rely on application of reusable changes to (i) minimise the efforts and (ii) maximise the efficiency of architectural evolution process. In recent years, the research state-of-the-art (in terms of patterns and styles) have mainly focused on the application of reuse knowledge to evolve architectures [Yskout 2012, Barnes 2014] with a clear lack of research on a systematic or empirical acquisition of such evolution-centric knowledge. The recent reviews of the research-state-of-the art [Breivold 2012, Williams 2010] suggest that the development of any solution that aims to enable knowledge-driven evolution must integrate the processes of knowledge acquisition and knowledge application in a unified framework that is lacking in the existing research.

In this research, we provide such an integrated framework called PatEvol - Pattern-driven Architecture Evolution. The first part of the proposed solution framework (or knowledge acquisition process) relies on post-mortem analysis of architecture evolution histories to discover change patterns to derive a pattern language for architecture evolution. The second part of the solution (or knowledge application process) aims at searching and selecting the appropriate patterns to promote pattern-based and reuse-driven architecture change implementation.

1.2 Implication of Change Reuse on Research and Practices for Architectural Evolution

To analyse the significance of reusability, we highlight the implications of change reuse on research and practices for ACSE. We refer to *research on architecture evolution* as academic initiatives proposing theories or innovative solutions to address ACSE with results evaluated usually in a (controlled experimentation using) lab-based environment [Cámara 2013]. In contrast, the *practices for architecture evolution* refer to maintenance, evolution and adaptation of architectures for industrial software reflecting real-world challenges and solutions to enable or enhance ACSE [Slyngstad 2008, Mohagheghi 2004]. In terms of academic research, we provide an in-depth analysis of research state-of-the-art for architectural evolution and adaptation in Chapter 3. In the remainder of this section, first we highlight the impacts and needs for change reuse in supporting evolution of industrial-scale software systems [Slyngstad 2008, Cámara 2013]. Secondly, some survey-based [Slyngstad 2008] and empirical studies [Clerc 2007] of architecture evolution reflects an insight about practitioners' (software designers' and architects') view on the role of reuse in ACSE. We also highlight a lack of generalisation for results across different projects and provide a general overview of the efficiency of architecture evolution process relative to the degree of reuse.

In the context of a traditional software development life-cycle, software design (blue-print before implementation) and evolution (changes after implementation) are regarded as two distinct phases [Fayad 1997, Garlan 2004]. However, an alternative interpretation of the software development process promotes maintenance and evolution as reuse-oriented software development [Basili 1990]. Such view implies that knowledge and expertise acquired by an organisation (or individuals) during a continuous maintenance and evolution can be extended and reused across different projects and for the future development of software systems. To the best of our knowledge, there is no evidence of any industrial research that supports architecture evolution-reuse in an industrial scale software or its architecture. There are only a few initiatives representing academic solutions and surveys based on analysis of industrial data for architecture evolution that are highlighted as below.

1.2.1 Practical Benefits for Reuse-driven Architecture Evolution

In [Cámara 2013], the authors supported reuse of adaptation policies to support dynamic adaptation of the architecture for an industrial system called Data Acquisition and Control Service (DCAS). The DCAS system is used to monitor and manage highly populated networks of devices in renewable energy production plants. This research demonstrates that reuse of recurring adaptation strategies and policies saves about **40%** of the efforts for architecture evolution compared to an ad-hoc and once-off implementation of adaptive changes. Also compared to the ad-hoc changes, reusable adaptation policies required less time for execution of architectural changes.

In other research [Mohagheghi 2004], the authors analysed change requests from four different releases of a large telecom system architecture developed by Ericsson over a period of three years. The research highlights that change reuse have resulted in a) an *increased maintainability* evaluated in cost of implementing architectural change scenarios, b) *improved testability*, c) *easier upgrades*, and also d) *increased performance*. In addition, the impact of software reuse; especially exploiting COTS (Commercial Off-The-Shelf) components is essential to enhance reuse of architectural components.

1.2.2 Change Reuse from Practitioners' Point-of-View

The subjective influence of an architect on software architecting process cannot be eliminated [Li 2012]. Therefore, it is vital to consider the practitioners' view on the role of reuse in ACSE through survey-based studies in industrial context. In an interesting study (The Architect's Mind-set) [Clerc 2007] the authors performed a survey-based analysis in the industry. The authors collected feedback on the importance of architectural knowledge that can be shared and reused to design, develop and evolve software architectures.

In another industrial survey [Slyngstad 2008], the authors investigate the risk management in software architecture evolution process based on feedback from **82** practitioners. The feedback suggest that *a lack of reuse patterns during design and maintenance of architectures results in poor integration of architecture changes into implementation process affecting architecture design negatively*. In contrast, the use of architectural patterns and styles [Gamma 2001, Shaw 2006] provide proven architectural solutions to enhance reusability and quality of architectural design.

1.2.3 Efficiency of Architecture Evolution Process Relative to Change Reuse

To generalise the efficiency measure associated with architecture evolution process (based on discussion in Section 1.2.1, 1.2.2), we utilise the ISO/IEC 9126 - 1 quality model [Jung 2004], an international standard for the evaluation of quality characteristics of a software product and solution [Jung 2004]. More specifically, based on ISO/IEC 9126 - 1; we analyse the quality characteristics of evolution process with factors including: efficiency, performance, maintainability (modifiability and testability).

Some empirical studies on analysing software change [Mohagheghi 2004] and more specifically architectural change [Slyngstad 2008] highlight that Efficiency (E) of architecture evolution process is directly proportional to a) factor of Reuse (R) in evolution and inversely proportional to the b) factors of Efforts (F) and c) Time (T) required to enable evolution as:

$$E = \frac{R}{F_{\varphi}+T}$$

The factor F is a measure of the effort required to support maintainability, testability, modifiability and increased performance. Please note that φ represents a constant as an initial upfront increase in effort (required for acquisition of reuse knowledge and expertise) that are used for future evolution to decrease effort [Clerc 2007]. The relation of evolution process efficiency in regard to the required efforts and time from [Slyngstad 2008, Mohagheghi 2004] is also confirmed with the findings in [Cámara 2013]. More specifically, the results in [Cámara 2013] suggest that in order to acquire reuse knowledge and expertise of dynamic adaptation there is a need for development of an adaptation knowledge-base that can reduce future efforts of change implementation based on reusable adaptation.

We present this finding from the industrial studies in Table 1.1 that highlight the associated benefits as well as necessary requirements to achieve evolution reuse. In Table 1.1, the symbols denote (– denotes a decrease), (++ denotes an increase). For example, in [Cámara 2013] the relative values are represented as a) adaptation efforts (–), are decreased, while b) testability (++) is increased. In this thesis, we focus on increasing the efficiency and reusability of architecture evolution by integrating reuse knowledge and expertise in the evolution process.

(–) A decrease of factor, (++) An increase of factor			
	Research Reference	Benefits of Evolution Reuse	Efforts to Achieve Reuse
Reuse of Evolution and Adaptation in Industrial Software	[Cámara 2013]	Adaptation Efforts (–), Time (–)	Adaptation Reuse Knowledge
	[Mohagheghi 2004]	Time (–), Cost (–), Maintainability (++) Testability (++), Upgrades (++)	Use of COTS to Achieve Architecture Evolution-Reuse
Survey-based Studies on Evolution Reuse in Industry	[Clerc 2007]	Quality (++)	N/A
	[Slyngstad 2008]	Quality (++), Maintainability(++), Time(++)	Pattern and Style-based Development of Architectures

Table 1.1: A Summary of the Benefits and Efforts Required for Reuse of Architecture Evolution.

1.3 Overview of the Research Challenges

In this section, we provide a brief overview of the relevant research challenges that also helps to outline the contributions of the proposed solution. We outline the central research challenge as:

How to enable a continuous acquisition of evolution-centric knowledge that can be reused to promote generic, off-the-shelf architecture evolution

The challenge also highlights that reuse knowledge for evolution is not limited to utilising the change patterns. Although the role of patterns in reuse is vital, knowledge represents a more diverse collection such as processes and activities that support evolution. Knowledge acquisition or extraction activities include: a) analysing architecture change representation to investigate recurring changes, b) discover change patterns, c) build-up a system-of-patterns as a formalised knowledge collection.

1.3.1 Identification of Knowledge Discovery Sources

A critical challenge of knowledge discovery lies with identification of knowledge sources. Knowledge source represents a transparent and centrally manageable repository of evolution traces that helps in investigating the historical view of evolution [Zimmermann 2005, Kagdi 2007]. In the PatEvol framework - as the proposed solution - we are primarily concerned with exploiting architecture change logs [ROS-Distributions 2010] as a sequential collection of changes that accumulate in the log over time.

1.3.2 Discovery of Evolution-Reuse Knowledge

The challenge concerns with a systematic and experimental analysis of the change logs to discover an implicit knowledge that is represented as architectural change instances. Knowledge discovery involves a multi-step change mining of logs that includes analysing change instances from logs to discover reusable and usage-determined operations and patterns. Knowledge discovery process must be automated with appropriate customisation to ensure the scalability of solution when the type and size of log data are complex and large for any manual analysis.

1.3.3 Representation and Specification of Evolution-Reuse Knowledge

After identification, we must provide a consistent representation of knowledge in order to facilitate knowledge reuse. In order to achieve this, at least we must support flexible storage, searching, selection and retrieval of knowledge during evolution. This requires to leverage the existing data storage methods to enable the development of a reuse knowledge collection.

1.3.4 Application of Evolution-Reuse Knowledge

Finally, we must support knowledge-driven evolution strategies that refer to a systematic mapping between the problem-solution views and to derive a change implementation mechanism [Le Goaer 2008]. As opposed to utilising ad-hoc and once-off change execution, the challenge lies with application of reusable change operationalisation and patterns to address architecture evolution.

1.4 Research Problems and Proposed Solution

The primary objective of this research is to enable reuse of evolution-centric knowledge and expertise to tackle recurring evolution in Component-Based Software Architecture (CBSA) [Szyperski 2002, van der Aalst 2002]. Research motivation (cf. Section 1.2) and an overview of relevant challenges (cf. Section 1.3) highlight the needs for a process-centric approach to enable ACSE. We outline the central hypothesis and research questions below.

1.4.1 Central Hypothesis

We formulate the central hypothesis for this research as:

A continuous investigation of architecture evolution histories enables the discovery of evolution-centric knowledge that can be applied to enable reuse of architectural changes and enhance the efficiency of architecture evolution process.

We propose a continuous discovery of evolution-reuse knowledge. Therefore, we aim to exploit architecture change logs [ROS-Distributions 2010, Yu 2009] - representing evolution histories [Zimmermann 2005, Kagdi 2007] - that provide a sequential collection of architectural changes that have been aggregating over time. A systematic investigation of change logs helps us to discover change operation types, operation dependencies and change patterns. The discovered patterns can be combined to derive a pattern language for architecture evolution. A formalised collection of change patterns in the language represents a structured knowledge about problem-solution mappings in a specific domain that is the evolution of CBSAs.

Analysis of the hypothesis suggests that the research problem that we aim to address can be sub-divided:

- How to conduct 'post-mortem' analysis of architecture evolution histories in order to discover architecture evolution-reuse knowledge.

- How discovered knowledge could be exploited to support reuse in architecture-centric software evolution.

1.4.2 Research Questions for the Thesis

In order to divide the central hypothesis into a set of related research problems, we outline the research questions as follows. Each of these research questions outline a central challenge and an individual aspect of the proposed solution.

- **Research Question 1** - How to model evolution histories that enable an experimental investigation of architecture change representation and its operationalisation?

  Primary Objective of this research question is the selection of an adequate notation or model representation for architectural changes from evolution histories [Kagdi 2007] (i.e., change logs.). Architecture evolution modelling is fundamental to investigating change logs in a formal and automated way. This question allows us to evaluate suitability and efficiency of the proposed modelling notation to discover evolution-reuse knowledge.

- **Research Question 2** - What methods and techniques could be exploited to discover reuse knowledge from architecture evolution histories?

  Primary Objective is the analysis and selection of available methodologies that allow a formal foundations for modelling, analysing and discovering evolution-centric information from architecture evolution histories. This questions allows us to evaluate the accuracy and efficiency of solution to discover reuse-knowledge from evolution histories in a (semi-) automated fashion.

- **Research Question 3** - How can discovered knowledge be represented and selected from evolution knowledge-base to facilitate its reuse?

  Primary Objective aims at representation of the discovered knowledge that enables its sharing and potential reuse whenever needs for architectural evolution arise. This questions allows us to evaluate the suitability of representation for discovered knowledge. Another important criteria is accuracy of selecting most appropriate knowledge artefacts during ACSE process.

- **Research Question 4** - What methods and techniques could be exploited to apply reuse knowledge to evolve software architectures?

  Primary Objective is focused on knowledge application. It requires utilising discovered knowledge from knowledge collection that facilitates knowledge reuse to guide evolution. This question allows us to evaluate the extent to which proposed methods and techniques support reuse of architecture evolution to enhance efficiency of evolution process.

Please note that RQ1 - RQ3 specifically focus on the knowledge acquisition process by identifying knowledge sources, knowledge discovery and its representation for reuse. RQ4 specifically aims at knowledge application process by utilising the discovered knowledge to address frequent problems of architectural evolution.

1.4.3 Solution Framework

Based on the hypothesis and questions, an overview of the proposed contribution as an integration of architecture change mining and change execution processes is illustrated in Figure 1.1. The solution aims to integrate evolution knowledge in architecture evolution process to enhance reusable change implementation in ACSE.

Framework Processes

The solution in Figure 1.1 is represented as an integration of two processes. These processes include *architecture change mining* to enable evolution history analysis (i.e., mining for reuse knowledge discovery). The second process highlights *architecture change execution* to implement changes in architectures (i.e., change execution for knowledge-driven evolution).

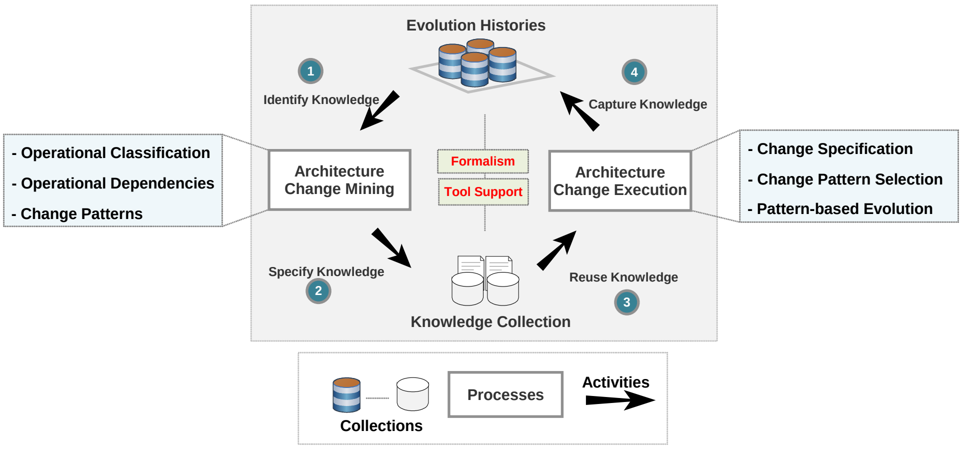


Figure 1.1: Overview of the Architecture Change Mining and Change Execution Processes.

Framework Activities

The framework activities consists of *knowledge identification*, *knowledge specification*, *knowledge reuse* and finally *knowledge capturing* to close the loop. The framework represents a cyclic approach to continuously discover and apply evolution-centric knowledge. Activities support the integration among the processes and highlights individual elements of the proposed solution framework.

Collections in the Framework

Collections are of two types with *evolution histories* [Zimmermann 2003, Kagdi 2007] representing the source of evolution-reuse knowledge. In addition, *knowledge collection* represents the repository that contains the identified knowledge that could reused. The role of formalism and tool support is complementary to ensure the scalability and customisation of the solution.

1.5 Research Contributions and Assumptions

We summarise the primary contributions as well as the assumptions for this research. Our research lies at the intersection of two distinct domains: i) *software repository mining* [Kagdi 2007] for architecture change analysis and ii) *software evolution* [Mens 2008] for architecture change execution. Based on an overview of the solution, first we introduce the proposed contributions (Contribution 1 - Contribution 4) and discuss the relevant assumptions (Assumption 1 - Assumption 3):

- **Contribution 1 - Evolution-centric Knowledge Discovery from Change Logs:** in order to discover evolution-reuse knowledge, we investigate architecture change representation from logs. More specifically, the evolution-centric knowledge from change logs is discovered in terms of different types of architecture a) *change operations* and b) *change patterns*. A change pattern provides a generic, first class abstraction - that could be operationalised and parametrised - to resolve recurring evolution problems in a specific domain (i.e., evolving architecture models). Change patterns follow the conventional philosophy behind the famous Gang-of-Four (GOF) design patterns [Gamma 2001]. However, in contrast to design aspects of software considered by GOF patterns, the proposed change patterns extend the reuse rationale to specifically address architecture evolution.

- **Contribution 2 - Algorithms for Mining Architecture Change Patterns from Logs:** we introduce the pattern discovery problem from change logs as a modular solution and present pattern discovery algorithms. Pattern discovery algorithms executed on architecture change logs enable automation along with appropriate user intervention and customisation of the pattern discovery process. Automation of pattern discovery process supports efficiency and accuracy for pattern mining from logs. Also, the scalability of pattern-discovery process beyond manual analysis is supported with a prototype G-Pride (Graph-based Pattern Identification) that enables automation and parametrised user intervention for pattern mining. In the context of patterns [Côté 2007, Yskout 2012] and style-based [Garlan 2009, Tamzalit 2010] evolution, the solution promotes a continuous discovery of new architecture change patterns from change logs over time[Yu 2009].

- **Assumption 1 - Availability of Log Data:** in order to discover the change patterns or the support for architecture change mining process in the PatEvol framework we need architecture change logs. Therefore, we assume that a change log must be available to investigate architectural changes and to discover patterns. Our assumption is based on the research that support post-mortem analysis of evolution histories to discover evolutionary knowledge that can be shared and reused to guide architectural evolution [Kagdi 2007, Zimmermann 2005].

  Moreover, according to the classification of change types (sequential vs parallel change [Buckley 2005]); the change log data consists of architectural changes that are captured as a sequence. We have assumed that parallel change operations (if any in the log) are represented as a sequence, where each of the change operations is executed one after the other (i.e., sequenced change log) [Williams 2010].

- **Contribution 3 - A Pattern Language for Architecture Evolution:** once we discover patterns, we go beyond the impact of individual patterns on architecture evolution to derive a change pattern language [4] that provides an interconnected system of patterns that enable reuse-driven and consistent evolution in Component-based Software Architectures (CBSAs). In a language context, interconnections represent possible relationships among patterns (such as variants or related patterns) in the language. We express evolution-reuse knowledge as a collection of interconnected patterns in the language with a vocabulary, grammar and pattern sequencing. We believe that by exploiting the vocabulary and grammar of a language as discussed in [Porter 2005], individual patterns can be formalised and interconnected to support reusable, off-the-shelf evolution expertise.

- **Assumption 2 - Existing Patterns in the Language**: the language development relies on the availability of a sufficient number of available patterns. Therefore, we assume that the

[4] In literature the terms *Pattern Language*, *Pattern Catalogue* or *Pattern Collection* are often used interchangeably [Zdun 2007]. However, we must maintain the technical distinction. A Pattern Catalogue or Pattern Collection may specify patterns formally or informally with no explicit relationships among patterns. A Pattern language must specify patterns formally and must support explicit relationships among patterns.

patterns in the language are sufficient (although not necessarily exhaustive) to support pattern-based evolution. New patterns can be discovered and continuously accommodated in the language. Our assumption is based on the fact that pattern discovery is a continuous process (by investigating change logs). As we acquire new data from different log sources we can execute the pattern discovery algorithms (on logs) to discover new patterns. If a pattern is needed that does not exist in the pattern language, then pattern-based architecture evolution is not supported. Instead of reusable patterns, more primitive changes as individual change operations are still supported.

- **Contribution 4 - Pattern Selection and Reuse in Evolution of Component-Based Software Architecture:** in order to promote reuse knowledge-driven evolution in CBSAs, we exploit pattern sequences from the language to support architecture change execution. It is vital to mention the pattern selection problem because it is a significant challenge for inexperienced developers or architects to search for and select the appropriate patterns from large collections [Kampffmeyer 2007]. With language-based formalism we exploit the Question-Option-Criteria (QOC) methodology [MacLean 1991] to address the pattern selection problem. The QOC methodology is adopted from design space analysis [Zdun 2007] to select the most appropriate pattern from the language collection by evaluating the forces and consequences of given patterns [MacLean 1995]. The patterns from the language could be selected and applied in a sequential fashion to support evolution.

- **Assumption 3: Application Domain of the Pattern Language** in the software architecture community, three of the well-established paradigms for architecture representation are object-oriented [Tu 2002], component-based [Szyperski 2002] and service-oriented [Erl 2009a] architectures. Structural evolution of component-based (also service component) architectural models is supported with the proposed pattern language. The patterns represent a generic and repeatable solution to recurring problems in a specific domain. Therefore, the application domain of the proposed pattern language is CBSAs and their evolution. The possibility of extending the proposed pattern language to other types of (non component-based) architectural models and their evolution is detailed in sub-sequent chapters.

1.6 Overview of the Thesis Chapters

The structural organisation of the thesis is presented in Figure 1.2. In the remainder of this section, we provide an overview of the role of each chapter and its outcome for in Table 1.2.

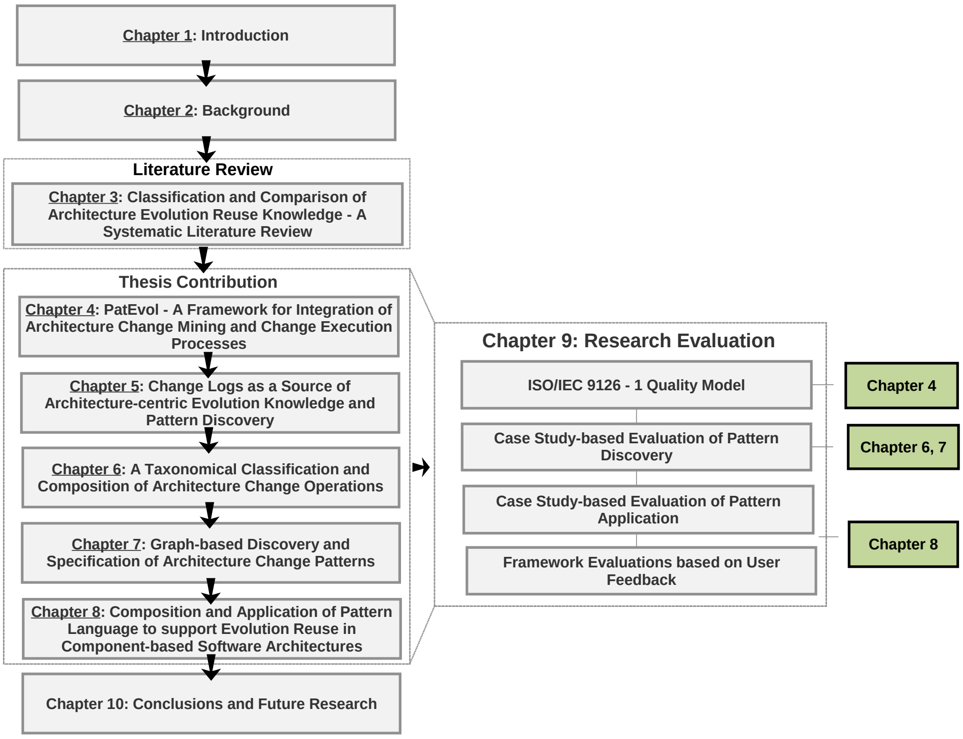

Chapter 1: Introduction
Chapter 2: Background
Literature Review
Chapter 3: Classification and Comparison of Architecture Evolution Reuse Knowledge - A Systematic Literature Review
Thesis Contribution
Chapter 4: PatEvol - A Framework for Integration of Architecture Change Mining and Change Execution Processes
Chapter 5: Change Logs as a Source of Architecture-centric Evolution Knowledge and Pattern Discovery
Chapter 6: A Taxonomical Classification and Composition of Architecture Change Operations
Chapter 7: Graph-based Discovery and Specification of Architecture Change Patterns
Chapter 8: Composition and Application of Pattern Language to support Evolution Reuse in Component-based Software Architectures
Chapter 10: Conclusions and Future Research
Chapter 9: Research Evaluation
ISO/IEC 9126 - 1 Quality Model
Case Study-based Evaluation of Pattern Discovery
Case Study-based Evaluation of Pattern Application
Framework Evaluations based on User Feedback
Chapter 4
Chapter 6, 7
Chapter 8

Figure 1.2: An Overview of the Thesis Organisation

- **Chapter 2:** after an overview of the research problems and proposed solution, we explain some of the fundamental concepts that provide background details on the role architecture evolution-reuse knowledge in the context of a) architecture knowledge and b) knowledge management in software engineering.

- **Chapter 3:** presents a critical review of research state-of-the-art in evolution-reuse knowledge for software architectures based on the finding of a systematic review. We present a systematic review of research supporting a) acquisition of b) application of reuse knowledge.

- **Chapter 4:** provides the solution overview in terms of a conceptual framework called PatEvol. The framework integrates architecture change mining and change execution to provide an iterative and process-centric approach to achieve evolution reuse.

- **Chapter 5:** discusses the role of architecture change logs as a transparent and centrally manageable repository consisting of a collection of change instances. We represent change instances from logs as an attributed graph for analysing change operationalisation and discovering change patterns.

- **Chapter 6:** is focused on analysing a taxonomy of architecture changes as represented in the logs. More specifically, we investigate change logs to analyse and classify change representation as a foundation for change pattern discovery.

- **Chapter 7:** details the discovery of change patterns from architecture change logs. We exploit the concepts of sequential pattern mining [Agrawal 1995] to identify recurring change instances from logs as change patterns. The discovered patterns are documented in a change pattern template to promote pattern-based reuse.

- **Chapter 8:** is the final chapter about the thesis contributions and details the composition and application of a change pattern language. The pattern language comprises of the pattern vocabulary, grammar and an interconnected sequence of patterns to support evolution.

- **Chapter 9:** is aimed at the evaluation of individual research activities and overall validation of the research hypothesis. We utilise the ISO/IEC 9126 - 1 [Jung 2004] model to evaluate the various aspects of the proposed solution. A case-study based evaluation is also complemented with prototype evaluation to measure process efficiency and adequacy of results.

- **Chapter 10:** concludes our research contribution in the context of identified research gaps. We overview the contributions, discuss potential limitations and validity threats along with the potential for future research.

Chapter	**Related Publication(s)**	**Outcome**
1	[Ahmad 2010]	*Hypothesis and Solution*
2	[Ahmad 2011]	*Thesis Background*
3	[Ahmad 2014d]	*Systematic Review Document*
4	[Ahmad 2013b, Ahmad 2012e]	*PatEvol Framework*
5	[Ahmad 2012b, Ahmad 2012c]	*Change Log Graph*
6	[Ahmad 2012b]	*Operational Classification*
7	[Ahmad 2012b, Ahmad 2013a]	*Architecture Change Patterns*
8	[Ahmad 2012a, Ahmad 2014c]	*Change Pattern Language*
9	N/A	*Research Validation*
10	[Ahmad 2012a, Ahmad 2012d]	*Conclusions and Outlook*

Table 1.2: A Mapping of the Related Publications to the Individual Chapters in the Thesis.

1.7 Chapter Summary

In summary, Chapter 1 provided the research motivation based on a quick overview of existing research and its limitations. Based on an overview of the needs for evolution reuse and identification of the research challenges, we outlined the central hypothesis that allowed us to identify the research questions. The role of individual research questions is vital in highlighting the requirements for the solution regarding research contributions and assumptions. A summary of the objectives and the outcome for the individual chapters in this thesis is presented in Table 1.2 that allows us discuss the literature review, research contributions and evaluation in subsequent chapters. The concepts and terminologies used in this chapter are used throughout the thesis.

Chapter 2

Background

Contents

2.1 Chapter Overview

The aim of this chapter is to provide background details about the foundational concepts and topics before the discussion of the technical contributions in subsequent chapters of the thesis. Therefore, the concepts and topics presented in this chapter are used throughout the remainder of this thesis and need an upfront explanation before technical details. We mainly focus on a theoretical background, however if a concept is of a practical relevance we also provide details about its (conceptual) modelling and necessary explanation about its implementation. For example, modelling software architecture as a graph we need to explain a) *What* are graph theoretical foundations to model architectures, b) *How* an architecture model is expressed as a graph and c) *Why* architectural descriptions are provided using graph modeling language [Brandes 2002a]. We use some running examples from case studies to clarify technical details in this chapter. Architecture evolution case studies [EBPPCaseStudy , 3-in-1 Phone System 1999, Rosa 2004] are presented in **Appendix B**.

2.2 Reuse Knowledge Management in Software Architectures

Knowledge management approaches [Bjørnson 2008] have been exploited across different dimensions in software engineering domain including requirements engineering [Hudlicka 1996], software architectures [Li 2012, Babar 2009], software testing [Wei 2007] and software documentation [Kiwelekar 2010] as illustrated in Figure 2.1. In software architectures, knowledge management represents discovering, representing and reusing knowledge and expertise to improve the architecting process [Li 2012]. In a general context of software engineering, first we highlight activities in *software architecture knowledge* that allows us to specifically focus on *architecture evolution-reuse knowledge* in Figure 2.1.

2.2.1 Architecture Knowledge Activities

Architecting a software system involves an extensive knowledge including but not limited to architectural design [Babar 2009], trade-off analysis [Garlan 2009], architectural documentation [Kiwelekar 2010] and evaluation [Li 2012]. This means, software architecting itself is a knowledge-

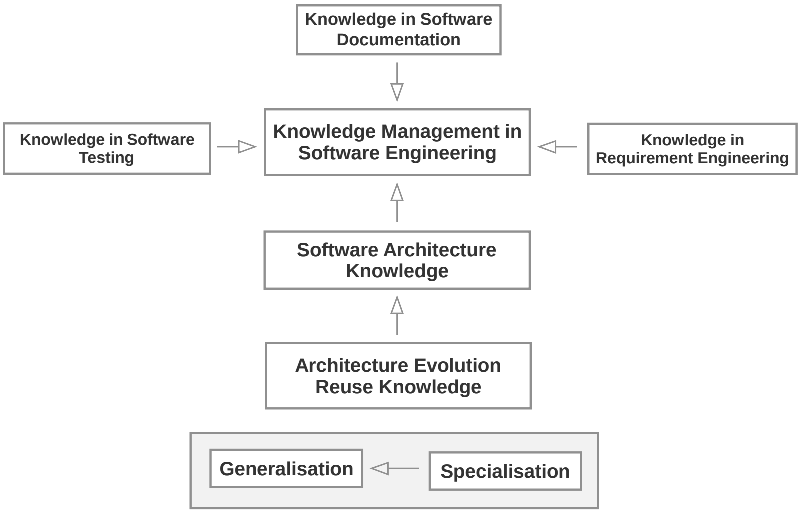

Knowledge in Software Documentation
Knowledge in Software Testing
Knowledge Management in Software Engineering
Knowledge in Requirement Engineering
Software Architecture Knowledge
Architecture Evolution Reuse Knowledge
Generalisation
Specialisation

Figure 2.1: Overview of Knowledge Management in Software Engineering and Architecture

intensive process that comprises of many sub-processes also known as architecture knowledge activities. A systematic mapping study of architecture knowledge [Li 2012] presents five generic architecting activities including: i) *Analysis*, ii) *Synthesis*, iii) *Evaluation*, iv) *Implementation* and finally v) *Maintenance and Evolution* of architectures.

In recent years, research [Breivold 2012] and practices [Cámara 2013] proposed solutions that specifically focused on applying reuse knowledge to guide architecture evolution process. Also, based on the findings of a systematic review in [Li 2012] and an overview of the architecting activities above, we conclude that architectural knowledge enables the application of concepts borrowed from knowledge management in software engineering [Bjørnson 2008] to specifically focus on guiding the software architecting process [Babar 2009] as illustrated in Figure 2.1. When considering the various activities in architecture knowledge, we are specifically interested in reuse of architecture evolution knowledge (Activity V). We position the concept of architecture evolution-reuse knowledge in the context of a) knowledge management in software engineering [Bjørnson 2008] and b) specifically as an activity of the architectural knowledge [Babar 2009].

2.2.2 Architecture Evolution-Reuse Knowledge

A systematic mapping of architecture knowledge research [Li 2012] suggests minimal evidence on application of reuse knowledge in maintenance and evolution of software architectures. Also, in Chapter 1 we discussed how evolution-reuse knowledge supports increasing efficiency and decreasing efforts of change implementation in ACSE process. Therefore, we believe that in a general context of architecture knowledge, there is a need to explicitly focus on classification and comparison of the existing body-of-research [Garlan 2009, Le Goaer 2008, Breivold 2012] that focuses on reuse knowledge to address recurring evolution problems in architectures. Also, the findings of our review (in Chapter 3) confirm the needs for: a) methodologies to discover evolution-reuse knowledge (also *knowledge acquisition*) that can be shared and reused and b)techniques that can exploit the discovered knowledge to promote reusable evolution (also *knowledge application*).

2.2.3 The Notion of Architecture Change Logs

In software evolution, the concept of change logs - maintaining a history of source-code level changes- is well established and have been exploited for an empirical investigation of source-code changes that have been implemented over-time [Kemerer 1999]. In comparison, there is a lack of research on maintaining and mining architecture change logs that abstract source code level changes with details of software architectural evolution (as addition or removal of architectural components and their connectors). In recent years, there is some research on analysing repositories (release histories [Wermelinger 2011] and change logs [ROS 2010]) that primarily focus on maintaining and analysing architecture-centric evolution of a software system.

Listing 2.1: An Example of the ROS Change Log

```
^^^^^^^^^^^^^^^^^^^^^^^^^^
Changelog for package foo
^^^^^^^^^^^^^^^^^^^^^^^^^^
0.3.4 (2013-04-09 16:36:55 -0700)
Released by: Sally <sally@example.com>
 ------------------------------------------
- Added thread safety node
- Replaced custom XML message with `TinyXML <http://www.grinninglizard.com/tinyxml/>`_.
```

ROS (Robot Operating System) change logs captures each individual change applied on ROS architectural elements with nodes (representing architectural components) and messaging (representing architectural connectors) among nodes. Therefore, the core architectural elements subject to change implementation are nodes and messages. Moreover, the change log also maintains the information about the person who applied the change as well as the date and time of change implementation. Listing 2.1 is an illustrative example of the ROS change log (partial representation) as below. Another example of architecture change logs is the release history of *Eclipse plugins* and their dependencies to analyse the structural evolution of the Eclipse [Wermelinger 2011]. The study analyses release histories by abstracting the source code files as Eclipse plugins (architectural components) and plugin dependencies (architectural connectors).

Based on the examples above, when we consider software architecture and architecture change log, different approaches may use different terminologies referring to the same concept. For example, in ROS the components are called nodes, whereas the connectors are referred to as messaging. In comparison, the Eclipse release history represent components plugins and their connectors as plugin dependencies. Also, in the examples above the repositories that capture and represent architectural changes are referred to as release history and change log. In our research, we use more well-known and widely used terminologies as architectural components and connectors whose changes are captured in architecture change logs.

2.3 Change Patterns as Elements of Architecture Evolution Reuse

In this section, first we highlight the role of architecture change patterns to enable evolution reuse. We also present a three-phase process that includes pattern identification, pattern specification and pattern instantiation activities during evolution. Finally, we discuss about architecture change pattern language [Alexander 1979, Goedicke 2002] that represents interconnected change patterns.

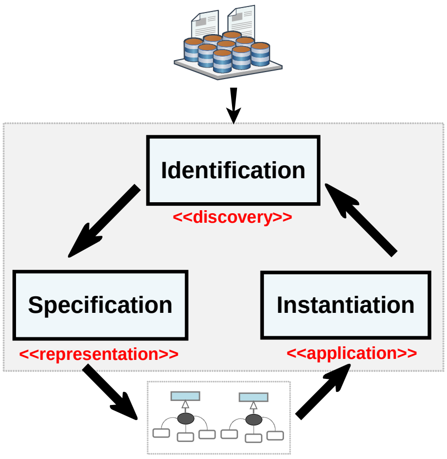


Figure 2.2: Overview of a Cyclic Process for Change Patterns in Architecture Evolution

2.3.1 A 3-step Process for Pattern-based Architecture Evolution

In the context of software architectures, change patterns [Yskout 2012, Côté 2007] promote reuse of change expertise with corrective, adaptive and perfective type changes [Williams 2010] to support design-time evolution [Côté 2007] and run-time adaptation [Gomaa 2010]. We further investigate the role of change patterns to enable evolution reuse through literature review in Chapter 3. Here we propose a cyclic process and present change patterns as a generic solution that can be a) identified as recurrent, b) specified once and c) instantiated multiple times to support change execution as in Figure 2.2.

Activity I - Pattern Identification

It aims at investigating the history of architectural changes to identify recurring change sequences as patterns that occur frequently over-time. The identification activity also refers to the pattern discovery that depends on the availability of established sources that facilitate an experimental discovery. In the context of architecture evolution analysis, available sources of pattern discovery are architecture *change logs* [ROS-Distributions 2010] and *version controls* [Tu 2002] that represent a transparent and centrally manageable repositories of architecture evolution history [Kagdi 2007, Gîrba 2006]. Change logs and version controls contain fine-grained traces of evolution data-sets that can be queried and searched for post-mortem analysis of evolution histories and to ultimately discover architecture change patterns [Javed 2013]. Details about pattern identification from architecture change logs are presented in Chapter 7.

Activity II - Pattern Specification

After identification, it is vital to provide a consistent (once-off) specification for a collection of identified change patterns that represent reuse knowledge about change implementation [Tamzalit 2010, Yskout 2012]. In pattern specification context, a pattern template [Harrison 2007] provides a structured document to promote patterns as a solution that can be retrieved from the template whenever the needs for pattern usage arises. The guidelines in [Clements 2003, Harrison 2007] provide comprehensive details to develop template for change patterns specification as presented in Table 7.4. We utilise the template in Table 7.4 for change pattern specification in Chapter 7.

Activity III - Pattern Instantiation

Finally instantiation utilises abstract specification of a pattern (from template) to instantiate multiple concrete instances of architecture change patterns. It promotes the concept of 'specify-once', 'instantiate-often' approach during architecture evolution. In a technical context, pattern instantiation enables mapping among the evolution problem to an appropriate pattern from template

Template Element	**Description of Element**
Pattern Description	
Name	Descriptive name to identify a given change patterns.
Intent	Primary intent of change pattern in the context of a given evolution scenario.
Problem	Details about the specific evolution problem that pattern solve.
Solution Example	Exemplified details and the context in which the pattern can be applied.
Context and Forces	
Constraints	Preconditions and post-conditions on source and evolved architecture model.
Change Operations	Required change operationalisation in order to apply changes using given pattern.
Architecture Model	Architectural descriptions before and after the application of change patterns.
Variants and Relations	
Variants	Variations in existing implementation of a given pattern.
Related Patterns	Relationships of the given pattern to other patterns in a pattern collection

Table 2.1: Template-based Specification of Architecture Change Patterns

to offer a generic reusable solution. The instantiated patterns and their corresponding change operations could again be captured in knowledge source that provide a continuously updated evolution-centric data for post-mortem analysis of architectural changes. We present pattern instantiation for architecture evolution in Chapter 8.

2.3.2 Composition of an Architecture Change Pattern Language

In software architecture research, the concept of pattern language is borrowed from theory of natural languages and the architecting experience (real-world buildings) by Christopher Alexander [Porter 2005, Alexander 1999]. Alexander's work also draws an explicit analogy between a pattern language to a natural language in the real world as both share the concepts of a vocabulary, grammar and language sequencing. Moreover, Porter's work on pattern language composition [Porter 2005] suggests that a pattern language provides the dynamics for generating (pattern) sequences, similar to the grammar of a natural language that provides the dynamics for generating sentences. For example, the target or application domain of a natural language is communication in real world, whereas change pattern language aims to solve recurring problems of architecture evolution. In the natural language, repeated words refer to patterns that are governed by language vocabulary and the sequencing of words. In addition, a natural language evolves over time by accommodating new words in the language. Therefore, we can exploit the foundational concepts of natural language in terms of its vocabulary and grammar to compose and evolve change pattern language by incorporating newly discovered patterns over time.

Pattern *language vocabulary* refers to the collection of identified instances of change patterns and their possible variants. Pattern *language grammar* specifies the structural composition of individual patterns and the rules that govern relationships among patterns in the language. Finally, pattern *language sequencing* defines an ordered sequence (following Alexander's theory[Alexander 1999, Alexander 1979]) of application among change patterns in the language. The growing research needs for pattern-languages are also highlighted with a dedicated series of conferences such as PLoP [1] and EuroPLoP [2]. In summary, the theory of pattern languages allows us to compose patterns that support an incremental change management [Goedicke 2002]. By incremental change management we mean: decomposing evolution process into a manageable set of evolution scenarios that can be addressed in a step-wise manner. We discuss composition and application of pattern language in Chapter 8.

2.3.3 Modelling of Pattern-based Architecture Evolution Activities

The process of pattern-driven architecture evolution requires modelling of the primary activities (from Section 2.3.1) that include *Pattern Identification*, *Pattern Specification* and *Pattern Instantiation* presented in Table 2.2. The identified objectives of modelling in Table 2.2 for each activity are quite distinct. For example, pattern identification aims at modelling architecture change instances to formalise change log analysis. In contrast, pattern specification requires meta-modelling of pattern language, while pattern instantiation requires pattern-driven architecture evolution. Such diverse modelling objectives for different activities makes it challenging to select the most appropriate formalism or notation that supports pattern-based architecture evolution. In the following we discuss some of the relevant formal approaches and then discuss our preferred approach to formalise pattern identification and pattern application alongside its benefits and limitations.

The Needs for Graph-based Mining of Sequential Patterns

Sequential patterns mining is viewed as a sub-domain of data mining to discover sequences of events or interests (e.g: sequence of web page traversals, sequences of chemical compounds) that occur frequently in a given data set. For example, one of the pioneering work on mining sequential patterns focused on discovery of frequent transactions by matching sequences of records in a customer transaction database [Agrawal 1995]. Since then there is tremendous growth of research with various approaches and algorithms to tackle the arising problems of sequential pattern mining in various domains [Mooney 2013]. This work also motivates our approach to mining patterns from sequence of architectural changes from architecture change logs. We select graph as a formalised model to represent the architecture change log and prefer a graph-based approach to

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

[2] EuroPLoP - European Conference on Pattern Languages of Programs: `http://www.europlop.net/`

mining the sequential change patterns [Huang 2003].

Over the years, some other notable approaches to pattern discovery included *database record matching* [Kum 2006], *XML-based template matching* [Leung 2005], *matrix* [Dong 2006] and *graph mining approaches* [Geng 2008, H. Tong 2007]. A detailed discussion of these approaches is beyond the scope here. Moreover, a comprehensive survey on the theoretical foundations, approaches and their algorithms for mining sequential patterns is presented in [Mooney 2013]. In comparison to other formalism for pattern mining such as XML or transaction matching [Leung 2005, Kum 2006], we prefer graph-based formalism for pattern discovery based on its well established theory and algorithms with more than three decade of research [Conte 2004]. We also highlight the benefits and limitations of graph-based formalism when compared to some other well-known techniques of pattern discovery. It is vital to mention that pattern discovery by means of analysing and matching database records [Kum 2006] is also considered an established approach for mining sequential patterns [Agrawal 1995]. Our preference for a graph-based approach is determined as graph theory provides support for both pattern discovery (graph mining) and pattern application for architecture evolution (graph transformation) - also discussed later in this Chapter. In summary, compared to other available approaches as discussed above graph theoretic approach provides the necessary theory, formalism and algorithmic support for both the *architecture change mining* and *architecture change execution* processes.

Benefits of Graph-based Pattern Discovery

- *Application of Sub-graph Mining Approaches for Pattern Identification* - An inherent beneïňĄt of graph based modelling of change log lies with the exploitation of well-established mathematical and algorithmic foundations for pattern discovery. More specifically, if architectural change could be modelled as a graph, sub-graph mining could be employed for pattern discovery in the change log graph. We utilise graph matching to investigate change representation and operational dependencies formulating foundations and to discover recurrent change sequences in the log. Analysing sequential operational compositions, we apply sub-graph mining [Jiang 2012] a formalised knowledge discovery technique - to identify recurring operationalisation that represent reusable, usage- determined change patterns.

- *Flexible Querying and Analysis* - Graph-based modelling also helps us with a flexible querying and ultimately the analysis of the log data when represented as a graph. Specifically, with a graph we could easily query (the graph nodes) for the scenarios such as ’retrieving all changes that remove architectural elements’ by simply scanning all the nodes (change operations) while by passing other data. Our evaluations (later in the thesis) show graph-based traversal of log data is more efficient than traditional file-based retrieval [Leung 2005].

- *Formal Representation of Architecture Changes* - In order to analyse and evaluate the evolution-centric information, a graph-based structure provide an appropriate data structure for modelling change instances as first class entities. More specifically, in the context of attributed graphs [Jiang 2012, Ehrig 2004] a graph node represents change operationalisation, while a node attribute represents the auxiliary data that represent the semantics (i.e.; what and why) of change. In graph-based representation, graph edges allow us to maintain a sequencing of change by means of interconnection among the graph nodes (change operations).

Limitations of Graph-based Pattern Discovery

After highlighting the benefits, we must also discuss the challenges or limitations of graph-based pattern mining approach. These limitations if not minimised or addressed properly could become threats to the validity of research.

- *Complexity of Pattern Mining* - One of the primary concerns with graph-based pattern mining lies with algorithmic efficiency of the process. Specifically, the graph-based modelling of change log data allows us to utilise the frequent sub-graph mining approach to discover recurring sequences (sub-graphs) as change patterns. By means of sub-graph mining, the nodes of sub- graph(s) (a.k.a. candidates) are iteratively matched to the nodes of a log graph to discover recurring sub-graphs (a.k.a. patterns). However, discovering patterns by matching and mining sub-graph is a complex problem that is known to be NP-complete [Conte 2004] and it is not known whether pattern discovery using graph mining is possible in a polynomial time. If the issue is not addressed it can lead to a significant and often exponential increase of computation time for pattern discovery.

- *Pre-processing of Log Data for Pattern Discovery* - In order to enable graph-based modelling, additional overhead that involves creation of change log also considered as a pre-processing for graph-based pattern discovery. In this pre-processing, the change operations and their sequence from log file are mapped to their corresponding nodes and edges in a change log graph. When the change log size is significant the manual efforts are impractical, error prone and time consuming, while there is a need for an automated creation of change log graph from log file.

2.3.4 UML vs Graph-based Modelling of Architecture Evolution Activities

It is also worth mentioning about the Unified Modelling Language 2.0 (UML) [Kandé 2000] that provides a standard and flexible modelling notation in terms of structural and behavioural diagrams. The UML 2.0 meta-model is defined with Meta Object Facility (MOF) [3] - an international standard ISO/IEC 19502:2005 - with four layers of abstraction to define structure and semantics of UML models. In addition, a summary of our comparative analysis (in Table 2.2) about formal notations to specify architectural description also suggests UML as one of the most prominent notations to represent architectural specifications and to evolve them using model transformation [Graaf 2007]. We analyse that UML 2.0 despite its flexibility, is not a suitable modelling notation for flexible representation of change instances from logs and to discover change patterns. In this scenario, we view graph-based models [Baresi 2006a, Carrière 1999, Bhattacharya 2012] as an alternative notation to represent distinct activities in evolution process.

Activity Modelling	**UML 2.0**	**Graph Models**
Activity 1 - **Pattern Identification**	No explicit notation to model change instances No explicit support to identify change patterns	Model Change Logs as an Attributed Graph Graph Mining to Identify Change Patterns
Activity 2 - **Pattern Specification**	Profile Diagram for Language Grammar Composite Structure for Pattern Template	Language Grammar as Attributed Graph Graph-based Pattern Template
Activity 3 - **Pattern Instantiation**	Architecture Model as Component Diagrams Architecture Evolution as Model Transformation	Architecture as Attributed Graph Evolution as Graph Transformation

Table 2.2: Comparison of UML 2.0 and Graph-based Formalism to Model Activities

Graph-based modelling has been successfully applied to perform analysis of software evolution [Bhattacharya 2012] and enabling architecture transformations [Baresi 2006a, Carrière 1999]. Therefore, in contrast to UML 2.0 modelling we believe graph-based models despite their general structure provide us with established formal foundations to specify all the activities of the framework in a flexible way. Therefore, in the context of overall evolution-centric activities and comparative analysis in Table 2.2, we claim that graph-based models provide an overall flexibility to represent all the activities in framework. In addition, we can exploit well established graph theoretical foundations - facilitating a) architecture change mining with *graph mining* algorithms [Jiang 2012] and b) architecture change execution with *graph transformation* [Baresi 2006a].

[3] OMG Meta Object Facility (MOF) Specification: `http://www.omg.org/mof/`

2.4 Graph Modelling for Change Mining and Change Execution

Based on the details of activity modelling and the comparison of UML 2.0 vs graph-based models, we overview different types of graph models following the basic notation from [Baresi 2002]. We explain graph-based modelling of architecture change mining and change execution processes.

2.4.1 Types of Graph Models

A graph G contain nodes N and edges E as a relation $G :=< N, E >$ based on graph models from [Baresi 2002]). The nodes in a graph represent the core entities of a graph that are inter-connected using edges. For example, an edge E connects two nodes N_1, N_2 as $E(N_1 \rightarrow N_2)$, where N_1 (source) and N_2 (target) are determined by the direction of the arrow in G. There exist a single or multiple edges between two graph nodes - determined by the type of the graph. Any node N in a graph G without any edge E (linked to N) is called an orphaned node. Therefore, each edge must have a source and a target node attached to it otherwise it is an orphan edge. In the following, using Figure 2.3, we summarise different types of graph models that help us to select the most appropriate type of graph-modelling for change mining and change execution processes.

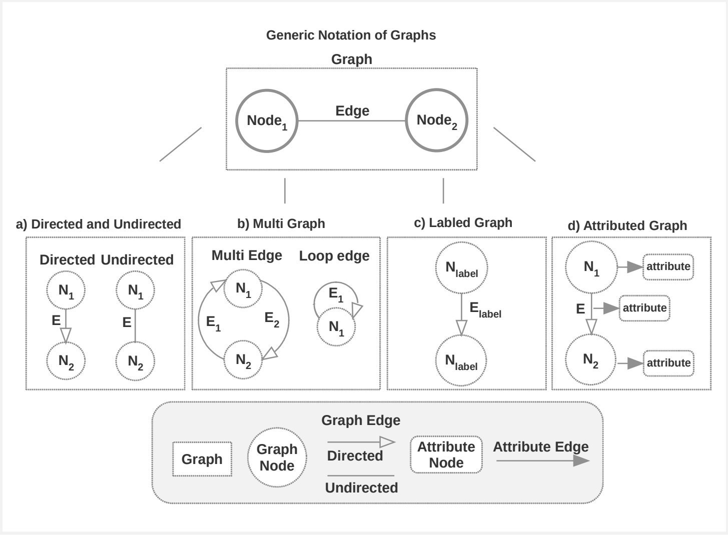


Figure 2.3: Overview of Different Graph Types

Directed and Undirected Graphs

As explained earlier, nodes of a graph are interconnected using edges. Graph edges can either be directed or undirected that represent a graph type as directed and undirected as presented in Figure 2.3 a). In an undirected graph the edges do not show any direction (from source to target node), for example edge $E(N_1, N_2) \equiv E(N_2, N_1)$, where $\equiv$ defines logical equivalence. On the contrary, we define a graph as directed when edges represent direction from source to target nodes. For example $E(N_1 \rightarrow N_2) \not\equiv E(N_2 \rightarrow N_1)$, where $\rightarrow$ represent source to target direction and $\not\equiv$ defines logical non-equivalence. In case of a directed graph edge $E(N_1 \rightarrow N_2)$ is considered directed from N_1 (source node) to N_2 (target node). In a mixed graph G (directed and undirected) some edges may be directed and some may be undirected.

Multi Graphs

In a graph G, there may exist multiple edges between two nodes of a graph $G :=< N_1, E_1, E_2, N_2 >$, such that $E_1(N_1 \rightarrow N_2)$ and $E_2(N_1 \rightarrow N_2)$ in Figure 2.3 b). In some other cases, it is possible for an edge to start and end on the same graph node $E_1(N_1 \rightarrow N_1)$ that is called a loop edge. A loop edge may be directed or it could also be undirected. A multi-graph is a special type of graph that allows multiple edges and loops among its nodes, not supported with normal graph types. However, the loops or multi-edges may or may not be permitted determined by the requirements/constraints explicitly specified on a graph.

Labelled Graphs

In a graph G it is common to have nodes and edges with some defined labels that represent (node-labelled, edge-labelled) graph as: $G_{labelled} :=< N1_{label}, E_{label}, N2_{label} >$ in Figure 2.3 c). In general, the term labelled graph only refers to a graph whose nodes are labelled, unless edge labelling is explicitly mentioned. Node and edge labelling enables embedding of extra information in a graph. For example, representing a country as a graph nodes contain labels as names of cities, while edge labelling represents the distance among two (nodes) cities $G_{country} := (N_{city1}, E_{distance_{(city1,city2)}}, N_{city2})$.

2.4.2 Attributed Graphs to Model Activities of Architecture Evolution

In a graph G, attributes can be associated to both the nodes and edges of a graph. In attributed graphs, nodes can be categorised among two types as graph nodes and the attribute nodes. Similarly, edges can be categorised as graph edges and the attribute edges. The attributes on nodes and edges can represent some property, data type etc. about the content represented by a graph node/edge. We expressed an attributed typed graph as: $G :=< G_A, N_G, E_G, N_A, E_{N_A}, E_{EA} >$.

$G \in Graph$ and G_A is graph attribute; $(N_G, N_A) \in Nodes$, where N_G is a graph node, N_A is an attribute node; $(E_G, E_{N_A}, E_{E_A}) \in Edges$, E_G is a graph edge, E_{N_A} is the edge of an attributed node, and E_{E_A} is edge of an attributed edge that is generalised in Figure 2.3 d).

We exploit attributed graphs [Ehrig 2004] to enable a formalised modelling of pattern-based evolution process activities. For illustrative purpose, in Figure 2.4 we provide an abstract view of an a) attributed typed graph and an b) attributed graph. More specifically, in Figure 2.4 a) the attributed type graph (ATG) represents graph-based meta-model with its instance model represented as an attributed graph (AG) that is typed over ATG in Figure 2.4 b). Nodes and edges in attributed graphs are typed over an attributed typed graph with attributed graph morphism: $AM : ATG \xleftarrow{TypedOver} AG$.

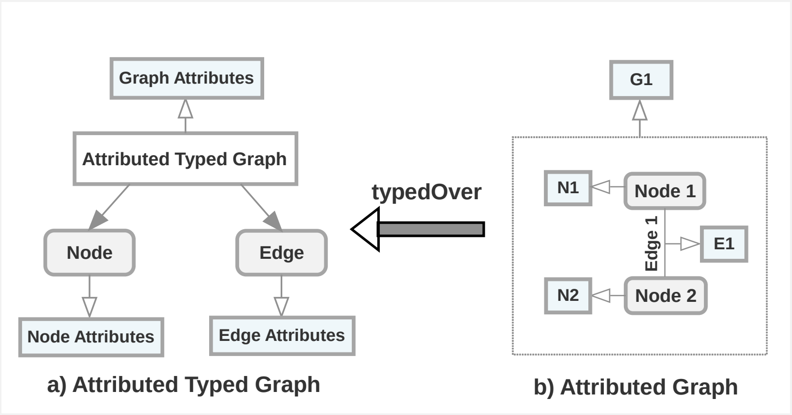


Figure 2.4: An Overview of a) Attributed Typed Graph and b) Attributed Graph - abstract view.

Attributed graph-based models have been proved successful for an effective modelling of the structure and behaviour of software architecture and their evolution [Baresi 2006b, Baresi 2002]. The benefits and limitations associated with the graph-based models (cf. Section 2.3.3) are also relevant to attributed graphs. In the following we discuss the primary motive for selecting the attributed graphs for change representation and pattern mining.

- *Benefit - Granularity of Representation for Change Log and Architecture Model* - Our motivation for exploiting attributed graph-based representation of the change log and also architecture model is based on a fine-granular representation of the data being analysed or transformed. By granularity of data we mean a detailed representation of the various type of information such as (i) the change operations, (ii) architecture model, as well as (iii) data about the date/-time, intent and scope of the architectural change. In comparison to the traditional graph models represented in Figure 2.3, the change log representation with an attributed graph in Figure 2.4 enables us to capture an individual change as a graph node while auxiliary information such as the source, date/time, intent and person who committed the change are also encapsulated in the attributes of the node. This means by utilising the attributed graphs we are able to distinguish between change operationalisation (nodes) and other auxiliary information (node attributes) not possible with ordinary graph models. Moreover, the graph edge represents a sequence among consecutive changes and can also contain extra information (if needed) in terms of the edge attribute. Further details about attributed graph-based representation of log data are provided in later in subsequent chapter once the change log data itself has been explained.

In the remainder of this section we focus on explaining how graph-based modelling from Figure 2.4 can be exploited to support the three activities for pattern-based architecture evolution process in Section 2.4.

2.4.3 Graph-based Modelling of Architectural Changes from Logs

In order to analyse representation of architecture changes from logs, we formalise individual change instances from log as a typed attributed to represent change operations on architecture elements - supporting change mining. Graph-based models have been successfully utilised for analysing software evolution [Bhattacharya 2012] and change investigation. A Change log is represented as an attributed graph with each graph node capturing an individual change and each graph edge represents a sequence among consecutive changes. Technical details about graph-based representation of architecture change logs are detailed in Chapter 5. We discuss the graph-based investigation of architecture change logs for change operation classification and pattern discovery in Chapter 6 and Chapter 7 respectively.

2.4.4 Graph-based Modelling and Transformation of Architecture Models

Finally, this section discusses the relevance of graph-based representation of the architecture model and exploiting graph transformation to support architecture evolution - supporting change execution. Specifically, we model the architectural structure as an attributed graph that provides formal representation with its node and edge attribution to express the hierarchical composition of architectural elements. By modelling architecture as a graph, an inherent benefit lies with exploiting graph-transformation to enable architecture evolution. It is logical to think that: *if graphs define* *the structure of architecture models, then graph transformation can be exploited to achieve transformation-driven architecture evolution in a formal way* [Baresi 2006b, Fahmy 2000, Baresi 2002]. In our solution, graph transformation of an architecture graph G allows us to utilise the graph-based formalism to create a modified or a target graph G_T out of an original graph G_S : $G_S \xrightarrow{transform} G_T$, where $G_S, G_T \in G$. In addition, we also discuss preserving the structural integrity of architecture model during evolution. Technical details about graph-based architecture evolution in Chapter 8.

2.5 Component-based Architectures and their Evolution

In 1992, Perry and Wolff proposed to build the foundations to study software architectures [Perry 1992] that entered its golden age almost a decade later as described by Shaw and Clements [Shaw 2006, Kruchten 2006]. In this era formal foundations, descriptions and modeling notations for architectural representations emerged that included but not limited to *object-oriented*, *component-based* and *service-driven models* [Erl 2009b, Stojanović 2005, Szyperski 2002] to develop and evolve software at higher abstraction levels. Traditionally, Object-oriented Software Engineering (OOSE); and more recently Component-based Software Engineering (CBSE) and Service-oriented Software Engineering (SOSE) promoted reuse of existing artifacts and entities to develop new software systems [Garlan 2009, Szyperski 2002, Erl 2009b]. The OOSE has established the foundations for CBSE as the component oriented technologies inherit the characteristics of object oriented technologies such as a collection of reusable objects that are modeled and utilised as software components.

Specifically, in CBSE the reuse of existing software artifacts - exploiting components - enable the development of new applications. The component-based engineering draws its inspiration from the success attained by other engineering disciplines that utilise the pre-built and standardised, off-the-shelf components. CBSE in general and CBSA in particular focus on componentising software representation, its development and evolution by incorporating various independent yet well-defined software pieces or artifacts as so-called components. Our preference for component-based architecture models and their evolution support in this thesis is based upon the facts that:

- We investigate change logs that represent architectural changes of CBSA (changes from object-oriented systems are investigated elsewhere [Tu 2002]). Therefore, architecture change analysis and discovered patterns could only be applied to a similar architecture that uses the component-connector notations for architectural modelling and evolution. Specifically speaking, the application domain of the discovered patterns is CBSAs and their evolution.

- In comparison to OOSE and SOSE, recently a comprehensive set of research and practices exploit component-based models for system development and evolution. We believe that that the theory and methodology of our approach is beneficial to support reusable evolution for software systems that exploits the notion of architectural components.

2.5.1 Modelling and Architecting with Component-based Models

The review of empirical research on CBSE [Tekumalla 2012] highlights that the research and practices using CBSA to model software systems can be categorised among two distinct dimensions: (i) formal foundations for architectural descriptions and (ii) frameworks or infrastructure for architecture-based implementations [Medvidovic 2000, Kandé 2000, van der Aalst 2002]. In case of formal foundations for CBSA, the prominent examples include *Architecture Description Languages* (ADLs) [Medvidovic 2000] and extensions in *Unified Modeling Language* (UML) [Kandé 2000] to specify architectural components their ports, connectors, and protocols to model software. In contrast, the solutions offering infrastructure for CBSA representation and its implementations included the commercial solutions in terms of middle-ware technologies and frameworks such as *Common Request Broker Architecture* (CORBA) from the Object Management Group (OMG), *Enterprise JavaBeans* from Sun Microsystem and the *Component Object Mode*l (COM) from Microsoft. In both these approaches the common objective remains same to exploit the architectural components their interfaces/ports and the coordination between components to represent the overall system and later concentrate on internal construction and workings of components. This means, CBSA provides abstraction on software models to shift the architects' and developers' focus from lines-of-code to components and their interconnection (with box and arrow) structure. Based on the discussion above and in the context of this thesis, we focus on the formal foundations and modelling of architectural components and their evolution.

The representation of our architecture model (Figure 2.5) and its formal specification (Table 2.3) is consistent with the definition of a component-based model in [Szyperski 2002] - having components as the computational units in the architecture that communicate with each other using component ports and are interconnected using the connectors. Once specified, we also discuss and exemplify some of the vital properties of CBSA that include but are not limited to component specification, their composition, interconnection and evolution. Earlier, we have detailed (in Section 2.3.3) that why in the context of the overall solution (change mining and change execution), we prefer graph-based modelling comparing to other available solution such as UML 2.0. Here, we only highlight why and how the graph-based modeling of CBSA is similar or distinct to component models in UML 2.0 (the component diagrams). It is vital to mention the role of the ADLs to formally specify and analyse the architecture models [Medvidovic 1997]. However, a recent survey of professionals on the industrial needs of the ADLs [Malavolta 2013] suggests that the ADLs emerging from academic research seem not to fulfill the industrial needs, even though they might have inspired the development of industrial ADLs in some ways. In contrast, the UML models and profiles have proved to be more effective means for modelling and analysing the architectures in a formal way. We must mention that any difference(s) between the graph-based model in Figure 2.5 and UML 2.0 components diagrams do not relate to the fundamental aspects, i.e; *what is a component or a connector*? Instead the difference lies with the preferred representation, i.e; *how to specify the component or a connector*.

- *Component Specification* - one of the primary limitation of our component-based model in Figure 2.5 is that it supports a limited type of components in an architecture that include the atomic and composite components (detailed later). In contrast, the UML 2.0 component diagram can exploit the UML stereotypes (represented with « ») to specify a number of customised components such as service, process, or implementation type of components. Such stereotype based specification allows more flexibility and customisation of components that is lacking in our architecture model. For example, the notation «service» can represent a stateless component and *«process»* represents a transaction-based component. The UML 2.0 represent the composite components as a *«subsystem»* while the atomic components that compose them are represented simply as *«component»*. In contrast, we use the nesting principle (or specifically graph node nesting) to specify an arbitrary number of atomic components nested inside a composite component. Such representation allows component composition by means of nesting atomic ones in the composite and component decomposition by removing the nesting among the atomic and composite components.

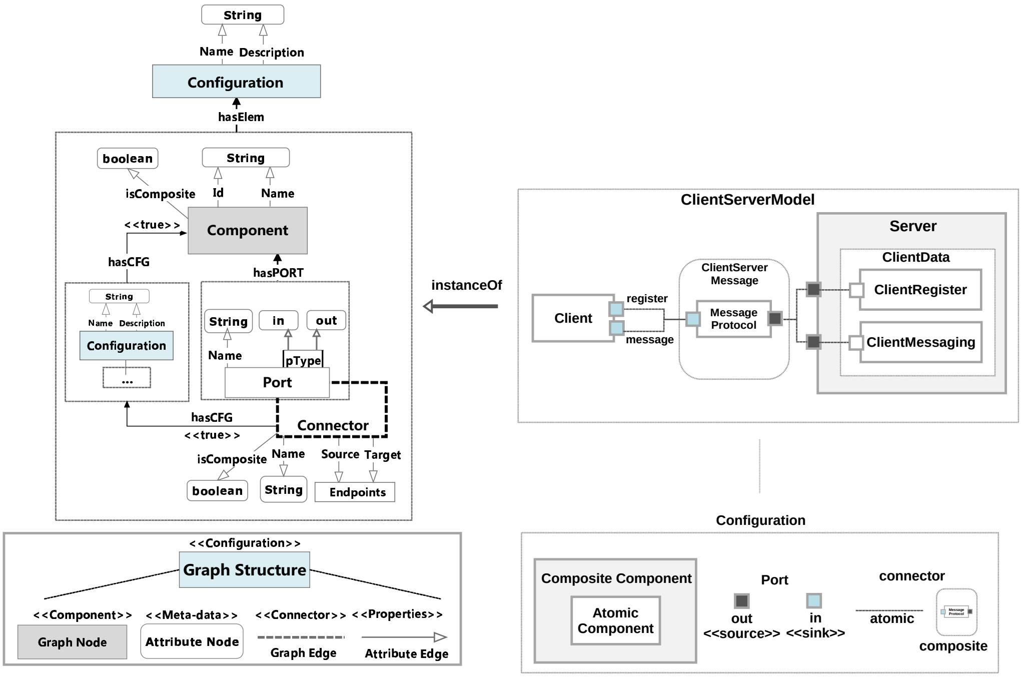
- *Connector Specification* - the UML 2.0 component diagram support two distinct types of connector namely the delegate and assembly connectors. Our graph-based model support both the delegation type interconnections. For example, in Figure 2.5 the component named Server delegates the functionality on its port to the port of its internal component named ClientRegister and ClientMesaging. In a concrete instance of architectural model the components are interconnected to each other as represented in Figure 2.5 b) that results in association and/or composition type interconnections among the components.

- *Component Association* represents the association type interconnection among two or more components. This means a provider component (on its ”out” port) provides some functionality to a requester component (on its ”in” port) via some component binding through connectors. The connector endpoint connected to the provider port is called the source, while endpoint to the requester port is called the target or the sink. For example, in Figure 2.5 b) the composite connector ClientServerMessage represents the association type interconnection between Client and Server.
- *Component Composition* represents the assembly or composition of one or more atomic components into a composite component. This is different to the association-type interconnection, the composite component contains sub-architecture (configuration among components and connectors) in itself. In addition, a mapping must exist among the ports of internal component to their composite component that allows external communication. For example, in Figure 2.5 b) the component Server is a composite that contains sub-architecture (in terms of architectural configuration ClientData) with a mapping among the composite Server and its child components ClientRegister and ClientMessaging.

The UML 2.0 component diagram only supports systems representation, whereas our graph-based modeling solution (cf. Section 2.3.3, Table 2.2) supports the analysis of architectural changes (graph mining) and implementation of architectural changes (graph-transformation).

2.5.2 Graph-based Modelling of Component-based Software Architectures

In component-based architectures, core architectural elements include Configurations among a set of Components and Connectors. A component is composed of one or more Ports to expose its functionality. The connector is composed of Endpoints that enable binding among the component port thus enabling component level inter-connections. In Figure 2.5, we introduce graph-based modelling to represent the structure of a component-based architecture model. We represent the architecture meta-model as an attributed typed graph (ATG) [Ehrig 2004, Brandes 2002b] in Figure 2.5 a). Architectural instances of a generic client server architecture is represented as an attributed graph (AG) that is typed over ATG in Figure 2.5 b). In this case, an attributed typed graph represents the architecture meta-model (Figure 2.5 a), while an attributed graph represents an instance model (Figure 2.5 b). Please note that the architectural description in Figure 2.5 b) is a partial representation of the architecture evolution case study from [Rosa 2004] that is detailed in **Appendix B**.



a) Typed Attributed Graph

b) Architecture Model (AG instanceOf ATG)

Figure 2.5: Attributed Typed Graph for Architecture Modelling

The nodes and edges of the attributed graph are typed over an attributed typed graph with the attributed graph morphism $A_M : AG \rightarrow ATG$. ATG provides a formalised approach with its node and edge attribution for modelling architectural elements as a hierarchical directed graph and preserves architectural composition relationships.

In this thesis, we only support structural evolution of the architecture and specify the structural descriptions of CBSA using the graph model as presented in Figure 2.5 and Table 2.3. In addition to the structural representation of architecture model in Figure 2.5 and its mapping to graph elements in Table 2.3, it is vital to discuss the fundamental characteristics based on the configuration of architecture models [Szyperski 2002, Medvidovic 2000].

2.5.3 Configuration of Architecture Model

As per the classification and comparison of architectural descriptions in [Medvidovic 2000], a configuration is a specific instance of the architecture model. More specifically, the architecture

Graph	Architecture Model	Description of Graph-based Architecture Model
$Arch_G$	Configurations	Typed attributed graph that represents the architectural configuration as a topology of components (attributed nodes) and connectors (attributed edges). **Examples** - *ClientData, ClientServerModel*
N_G	Component	Attributed graph node that represents the computational element and data store in terms of atomic and composite components in the architecture
	Atomic Component	A component with no internal configuration/sub-architecture. **Example** - *Client, ClientRegister, ClientMessaging*
	Composite Component	A component that has internal configuration/sub-architecture. It is represented as a nested graph inside the node. **Example** - *Server* with child components *ClientRegister*, *Client Messaging*
E_G	Connector	Attributed graph edge as atomic or composite connector among components.
	Atomic Connector	A connector with no internal configurations/sub-architectures **Example** - *register, message*
	Composite Connector	A connector that has internal configurations/sub-architecture. It represents a nested graph inside the edge. **Example** - *ClientServerMessage*
N_A	(Node-Edge) Attribute	Attributes corresponding to the nodes (N_G) and edges (E_G) in ($Arch_G$).
	Meta-data Attribute	Data labelling and additional information for graph nodes and edges **Example** - *Component Name, Description, Id, isComposite* etc.
	Element Attribute	Attribute representing Component Ports and Connector Endpoints
	Component Port Connector Endpoint	Inward (“*in*″) or Outward (“*out*″) points for Component inter-connections Provide connector bindings among particular “*in*″ port to a particular “*out*″ port
E_{NA}	Component Attribute Edge	Node attribute edge that connects the graph nodes to its attributed nodes.
E_{EA}	Connector Attribute Edge	Edge attribute edge that connects the graph edges to its attributed nodes.

Table 2.3: Mapping Graph Elements to the Architecture Model

model represents an abstract description of the necessary elements in terms of the architectural components and connectors. In contrast, a configuration is a concrete instance of the architecture model that addresses: what is the role of an individual component and how a collection of the components are interconnected to realise the architecture. Moreover, the configuration determines that necessary components and connectors exist, the component ports match, connector endpoint bind components, and the combined model results in a desired architectural structure.

2.5.4 Architecture Descriptions with Graph Modeling Language

In order to provide concrete description for the architectural graph $Arch_G$ in Table 2.3 we utilise the Graph Modeling Language (GraphML) [Brandes 2002a] that provides us with a comprehensive and easy-to-use file format for XML-based graph representation. In Listing 2.2, we represent the architecture elements from Figure 2.5 as a GraphML document. GML notations support:

1. **Header** in terms of an XML Schema reference that provides means to validate the topology of graph elements that are represented as a graphml (.GML) format (Line 02 - 06 Listing 2.2).

2. **Topology** a graph is represented in GraphML $< graphml >$ (Line 02 - 43) format by a $< graph >$ (Line 07 - 42) element. The $< graphml >$ element can contain any number of $< graph >$s. The nodes of a graph are represented by a list of $< node >$ (Line 11 - 41)

elements. Each node must have an id attribute. The edge (Line 35 - 40) set is represented by a list of $< edge >$ elements. Details on graphml format are presented in [Brandes 2002b].

Possible Extensions

We model the necessary set of first-class constructs for architecture elements including configurations among a set of components that have ports and connectors that have endpoints. GraphML provides extensions in the graph model through attributed tags or schema extensions. For example in the case of adding a set of operations on component ports, the new attribute $< operation >$ can be added as $< key\ id = “operation'', for = “port'', attr.name = “operationr'', attr.type = “string'' >< /key >$. Similarly the attributes like $< returntype >$ and $< paramlist >$ for corresponding operations can be added, the scope of which is beyond this work.

Listing 2.2: GraphML-based Model for Client Server System from Figure 2.5 (partial architecture)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <graphml xmlns = "http://graphml.graphdrawing.org/xmlns"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
5           http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
6     <!-- ... Outer Graph as Configuration...  -->
7     <graph id="ClientServerModel" edgedefault="directed">
8        <data key="ID"> Configuration_CS</data>
9        <data key="Description"> Configuration of Server Components </data>
10    <!-- ... Out Graph Node as Composite Component ...  -->
11          <node id="1">
12             <data key="Name">Server</data>
13    <!-- ... Component Port...  -->
14                   <port name = "port_RegisterClient">
15                     <data key="Direction"> "out" </data>
16                   </port>
17                     <port name = "port_ClientMessaging">
18                       <data key="Direction"> "out" </data>
19                      </port>
20                    <data key="isComposite"> true </data>
21    <!-- ... Inner Graph as Internal Configuration of Composite...  -->
22               <graph id="ClientData" edgedefault="directed">
23                           <data key="ID">Configuration_CD</data>
24                          <data key="Description">Internal Configuration </data>
25    <!-- ... Atomic Component (ClientRegister) in Composite Comonent (Server)...  -->
26                        <node id="2">
27                           <data key="Name">ClientRegister</data>
28                              <port name = "port_getClientRegister">
```

```
                                        <data key="Direction"> "in" </data>
                                   </port>
                                <data key="isComposite"> false </data>
                               </node>
                   </graph>
  <!-- ... Component Connector... -->
                    <hyperedge>
                       <data key="Name">Connector</data>
  <!-- ... Connector Endpoints... -->
                        <endpoint node="2" port=" port_getClientRegister "/>
                          <endpoint node="1" port=" port_RegisterClient ">
                      </hyperedge>
                ния</node>
   </graph>
</graphml>
```

2.6 Chapter Summary

In this chapter, we provide the background details about some of the fundamental concepts and terminologies that are used in the remainder of this thesis. First of all, we explain knowledge management in software architectures that allow us to discuss architectural knowledge and the needs for acquisition and application of architecture evolution-reuse knowledge.

We highlight the role of change patterns as a generic, reusable solution to recurring architecture evolution problems. During evolution, change patterns can be identified as recurrent solutions that can be specified once and instantiated multiple times to enable reusable change execution.

We prefer attributed graph based modelling for different activities in pattern-based evolution. We propose to represent architectural change instances from logs as a graph with graph mining for pattern discovery (*identification activity*) and represent the discovered patterns in a pattern template (*specification activity*). Finally, we model a component-based architecture model as an attributed graph. Graph-based modelling of architecture allows us to exploit graph transformation on architecture model to enable pattern-driven architecture evolution (*instantiation activity*).

Chapter 3

Classification and Comparison of Architecture Evolution-Reuse Knowledge - A Systematic Review

Contents

3.1 Overview of Systematic Literature Review

After presenting the research challenges and thesis background, in this chapter we provide a critical review of the existing research that addresses methods and techniques to support application and acquisition of architecture evolution reuse knowledge. In this review, we aim to *identify*, taxonomically *classify* and systematically *compare* the existing research focused on enabling or enhancing change reuse in architecture-centric software evolution. We conducted a Systematic Literature Review (*SLR*) [Brereton 2007] of 30 qualitatively selected studies, published from 1999 to 2012. The progress of architecture-centric evolution reuse research [S9, S7][1], [S2, S1] is reflected over more than a decade starting in 2001 [van der Hoek 2001]. However, we did not find any evidence that systematically synthesises the collective impact of existing literature on evolution reuse. As highlighted in Chapter 2, in the general context of architecture knowledge (AK) there is a need to explicitly classify and compare research on reuse knowledge to address recurring evolution in software architectures [S1, S2, S9, S17]. To carry out this review, we followed the guidelines in [Brereton 2007] to conduct a systematic literature review of evolution reuse in software architectures. The objective of this review is to:

Systematically identify and classify the available evidence about evolution-reuse in software architectures and provide a holistic comparison to analyse the potential and limitations of existing research.

In addition to the review results in this chapter, we present a) methodological details, b) qualitative analysis of the selected literature and c) threats to the internal and external validity of the SLR in **Appendix A**.

[1] Please note that, in this chapter only the notation $[S_N]$ (N is a number) represents a reference to studies included in the SLR. The notation also maintains a distinction between the bibliography and list of selected studies for SLR provided at the end of this chapter as: *List of Studies Selected for Systematic Review*.

3.2 Secondary Studies on Software Architecture Evolution

In this section, we summarise the existing SLRs (in Section 3.2.1) and survey-based studies (in Section 3.2.2) addressing architecture evolution in Table 3.1 to justify the needs and scope for this review. In recent years, SLRs [Brereton 2007] have focused on evolvability analysis [Breivold 2012], change characterisation [Williams 2010], classification and comparison [Breivold 2012] of ACSE. In contrasts to the existing systematic reviews on ACSE [Williams 2010, Breivold 2012, Li 2012], this SLR specifically focuses on a taxonomical classification and comparison of research that supports evolution reuse in architectures.

3.2.1 Systematic Literature Reviews of Software Architecture Evolution

Review of Architecture Change Characterisation

The systematic review (study reference [Williams 2010] in Table 3.1) investigated a total of 130 peer-reviewed studies - published from 1976 to 2008 - to characterise design-time and runtime evolution as corrective, perfective, adaptive and preventive type changes in architectures. The SLR [Williams 2010] proposed a comprehensive change characterisation scheme to distinguish and characterise software architecture changes and change impact analysis. The scheme works as a decision tree to provide support for system developers to assess the impact and feasibility of desired changes.

Review of Architecture Evolvability Analysis

The systematic review ([Breivold 2012] in Table 3.1) investigated 82 peer-reviewed studies - published from 1992 to 2012 - focused on design-time evolution of software architectures. The SLR [Breivold 2012] is focused on analysing the evolvability of a software architecture. The primary objective of this review is to provide an overview of existing approaches for analysing and improving software architecture evolution and to identify factors influencing architectural evolvability.

Study Type	Study Reference	Study Focus	Publication Year	Total Reviewed	Years
Systematic Literature	[Williams 2010]	Change Characterisation	2010	130	1976 - 2008
	[Breivold 2012]	Evolvability Analysis.	2011	82	1992 - 2010
	Review in this Thesis	Reuse-Driven Evolution	2014	30	1999 - 2012
Surveys	[Bradbury 2004]	Dynamic Evolution	2004	14	1992 - 2002
Mapping Studies	[Li 2012]	Architecture Knowledge	2013	55	2000 - 2011

Table 3.1: A Summary of the Secondary Studies on ACSE.

3.2.2 Survey-based and Taxonomic Studies on Architecture Evolution

Survey of Self-Management in Dynamic Software

A survey-based study ([Bradbury 2004] in Table 3.1) reviews a total of 14 studies - published from 1992 to 2002 - focused on runtime evolution of software architectures. This survey synthesises formal specifications for dynamic adaptation of software architectures. The authors present a set of classification criteria for the comparison of dynamic software architectures based on the types, processes and infrastructure for dynamic adaptation of software architectures.

Mapping Study on Knowledge-based Approaches in Software Architectures

A mapping study ([Li 2012] in Table 3.1) provides a systematic map of research on knowledge-based approaches in software architecture based on 55 peer reviewed studies - published from 2000 to 2011. The mapping study [Li 2012] identifies gaps in the application of knowledge-based approaches to five architecting activities that include architectural analysis, synthesis, evaluation, implementation, along with maintenance and evolution. The study shows an increasing interest in the application of knowledge-based approaches in software architecture with only 5/55 studies on architectural knowledge for maintenance and evolution.

Industrial Survey and Taxonomic Study on Architecture Evolution

In [Stammel 2011], the authors provide an overview of various approaches evaluated based on real-world industrial scenarios on the evolution of sustainable systems. This study targets practitioners because it is more general and is a live document based on a growing number of experience reports. Slyngstad et al. [Slyngstad 2008] performed a survey among software architects from software industry in order to capture a more complete picture of risk and management issues in software architecture evolution. Although not directly related to the ACSE, some taxonomies of software change [Buckley 2005, Chapin 2001] try to answer the questions like *why*, *how*, *what*, *when* and *where* aspects of software evolution that have acted as a guideline for us to define the comparison attributes in this review discussed in subsequent sections of the chapter.

3.2.3 A Systematic Review of Architecture Evolution Reuse Knowledge

Our review in this chapter of the thesis (as highlighted in Table 3.1) is focused on a systematic identification, classification and comparison of the existing research that supports application and acquisition of reuse knowledge to support ACSE. In contrast to the mapping study on AK [Li 2012] that identifies only 5 studies supporting design-time maintenance and evolution, our SLR is comprised of 30 studies published from 1999 to 2012 and is focused on both design-time and runtime evolution. As presented in Table 3.1, the proposed SLR complements the existing body of secondary studies on ACSE [Williams 2010, Breivold 2012, Jamshidi 2013b]. Given the importance of reuse in ACSE, it exclusively focuses on classification and comparison of evolution reuse knowledge.

In order ensure that a similar review has not been performed, we searched the Compendex, IEEE Xplore, ACM and Google Scholar digital libraries (on 23/10/2012) with the search string provided in **Appendix A**. None of the retrieved publications were related to any of our research questions detailed (Section 3.3). Considering the importance of reuse in ACSE [Breivold 2012] and the relative maturity of architecture knowledge (AK) research [Babar 2009, Li 2012], a consolidation of existing evidence about application and acquisition of reuse knowledge to support ACSE is timely.

3.3 Research Methodology for Systematic Literature Review

In contrast to a non-structured review process, a systematic literature review reduces bias and follows a precise and rigorous sequence of methodological steps [Zhang 2012, Brereton 2007]. More specifically, an SLR relies on a well-defined and evaluated review protocol to extract, analyse and document the results as illustrated in Figure 3.1. We adopted the guidelines in [Brereton 2007] for SLRs with a three step review process that includes: *Planning*, *Conducting* and *Documenting*. The review is complemented by an external evaluation of the outcome of each step, as illustrated in Figure 3.1. We also provide an explicit taxonomical classification of the reviewed studies. This is the foundation for a comparative analysis of studies based on our defined comparison attributes that are also subject to external evaluation prior to results reporting in this chapter.

Based on a three step process in Figure 3.1, the extended details of the definition and evaluation of the protocol (steps for *planning*, *conducting* and *documenting*) of systematic review are presented in **Appendix A**. In the reminder of this section, first we outline the **systematic review questions** that drive **literature search** followed by the **extraction and synthesis of the results** and finally **classifying and documenting the results** as in Figure 3.1.

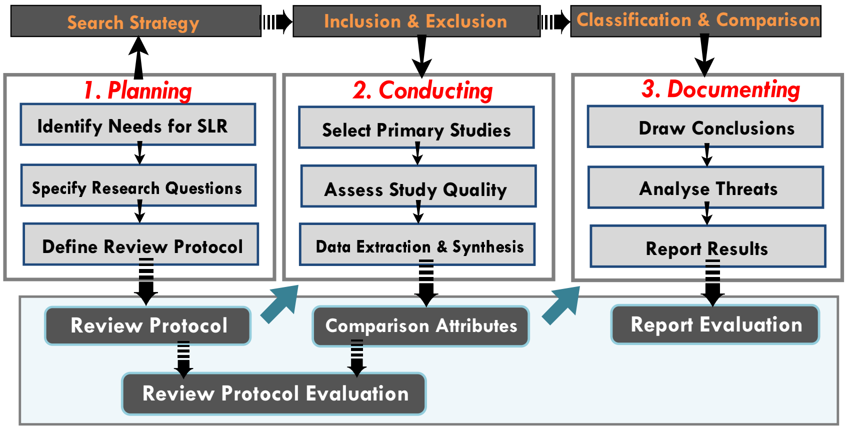


Figure 3.1: SLR Process for Classification and Comparison of Reuse-Knowledge in ACSE.

3.3.1 Research Questions for Systematic Review

The systematic questions are based on our motivation to conduct the SLR, i.e., the answers provide us with an evidence-based overview of the definition, application and acquisition of reuse knowledge to support ACSE methods and techniques. We define three systematic review questions that represent the foundation for deriving the search strategy for literature extraction. The motivation outlines the primary objective of investigation for each question. A comparative analysis allows us to analyse the collective impact of research, represented in terms of comparison attributes (in Table 3.2) for Systematic Review Questions (SR-Q) as below.

- **Systematic Review Question 1 -** *How evolution reuse knowledge is defined, classified and expressed in existing literature to enable architecture-based software change management?*

  Motivation: To understand the existing classification and representation of architecture evolution reuse knowledge and a detailed comparison of solutions to enable ACSE.

- **Systematic Review Question 2 -** *What are the existing methodologies and techniques that support application of reuse knowledge to evolve software architectures?*

  Motivation: To identify and compare existing solutions that support an explicit reuse of change implementation mechanisms to enable design-time evolution and run-time adaptations in architectures.

- **Systematic Review Question 3 -** *What empirical approaches are employed to discover evolution reuse knowledge?*

Motivation: To investigate and compare the available support for empirical acquisition/discovery of reuse knowledge and expertise that can be shared to guide architecture evolution.

In the remainder of this chapter, we refer to the systematic review questions simply as research questions or SR-Q.

3.3.2 Extracting and Synthesising Review Data

In order to record the extracted data from selected studies, we followed [Zhang 2012, Brereton 2007] and designed a structured format as presented in Table 3.2. The format in Table 3.2 records the results as: a) *generic and documentation specific* data items, and b) *comparison attributes* for a collective and comparative analysis of research and to answer SR-Q1 - SR-Q3.

Comparison attributes (CA1 - CA12 in Table 3.2) are the smallest unit of data that we extracted from the literature for comparison purposes and shared for external evaluation. These attributes provide the base for follow up syntheses, that is mainly classification and comparison of claims and their supporting evidence of evolution reuse detailed in this chapter. Due to time constraints of external reviewers, instead of reading through detailed results (in Section 3.4, 3.5, 3.6), they examined a summary of results and comparative analysis to suggest appropriate adjustments and refinements for documentation of results. These data were extracted by locating the evidences of each item in selected studies.

3.3.3 Classifying and Documenting the Results

To discuss the results, first we need to provide a conceptual framework to systematically present the existing literature and to identify the required steps that enable ACSE. With the help of a framework we can organise the reviewed studies in terms of framework processes and activities that support application and acquisition of evolution reuse knowledge.

3.3.4 A Framework to Classify Evolution Reuse Knowledge Research

Method engineering [Brinkkemper 1996] enables us to reuse the existing concepts from existing methods (frameworks, models or solutions) to develop new methods by reusing existing methodologies with reduced efforts and time to derive or develop new solutions. More specifically, during the architecture change mining process in the REVOLVE framework we exploit the knowledge discovery concepts from the ADM (Architecture Driven Modernization) [Ulrich 2010] model

ID	**ID**	**Objective**
Generic and Documentation Specific Data		
1	*Study ID*	Unique Identity of Study
2	Bibliography	List of Authors Year of Publication Source of Publication [*Book* or *Journal* of *Conference* or *Workshop* or *Other*]
3	*Focus of Study*	Theme, Concepts, Motivation clearly presented? [*Yes* or *No*]
4	*Research Method*	[Design and Evaluation or Case Study or Survey or Experiments or *Other*]
5	*Research Problem*	Research Challenges or Problems Reported
6 7	*Proposed Solution* *Application Context*	Solution to Address Research Challenges or Problems Context and application domain: [*Academic* or *Industrial* or *Both* or *Other*]
8	*Limitations*	Constraints, Limitations, Future research clearly stated? [*Yes* or *No*]
9	*Related Research*	Positioning and Novelty of the research
10	*Future Dimensions*	Implications on Future Research or Ideas clearly stated? [*Yes* or *No*]
Comparison Attributes for SR-Q1 and SR-Q2		
CA1	*Knowledge Support*	Solutions to support reuse-knowledge in ACSE.
CA2	*Type of Change*	[*Adaptive* or *Perfective* or *Corrective* or *Preventive*]
CA3	*Time of Change*	[*Design-time* or *Runtime*]
CA4	*Means of Change*	Type of Operational Support to implement change
CA5	*Formalism Support*	Application of a specific formal approaches in modelling,
CA6	*Architecture Descriptions*	[*UML* or *ADL* or *Graph Models* or *State Transition* or *Other*]
Comparison Attributes for SR-Q1 and SR-Q3		
CA7	*Knowledge Source*	The type of collection - real data set for change instances
CA8	*Type of Analysis*	Type of analysis to discover evolutionary knowledge
CA9	*Type of Formalism*	Type of formalised methods and for empirical discovery
CA10	*Time of Discovery*	[*Run-time Extraction* or *Off-line Mining* or Other]
Comparison Attributes for both SR-Q1, SR-Q2 and SR-Q3		
CA11	*Tool Support*	Automation support for reuse-driven evolution [*Yes* or *No*]
CA12	*Evaluation Method*	[*Design and Evaluation* or *Case Study* or *Survey* or *Experiments*]

Table 3.2: A Summary of the Extracted Data and Comparison Attributes.

for acquisition of evolutionary knowledge from architecture evolution histories. Moreover, the discovered knowledge can be shared and reused as in the MAPE-K (Model Analyze Plan Execute-Knowledge) [Ganek 2003] framework to analyse, plan and execute architectural evolution and adaptation.

The REVOLVE framework in Figure 3.2 along with the presentation of its processes, activities and their corresponding studies in Table 3.3 is beneficial for ACSE researchers and practitioners. The framework assist ACSE researchers with quick identification of relevant studies. A systematic presentation of existing research provides a foundational body of knowledge to develop theory and solutions, analyse research implications and to establish future dimensions. In addition, the framework can be beneficial for practitioners interested in understanding the methods and solutions with formalism and tool support to model, analyse, and implement evolution reuse in software architectures. The framework provides an aggregated representation of existing literature. The results will later highlight a lack of solutions that integrate the concept of empirical knowledge acquisition to guide evolution with reuse knowledge application.

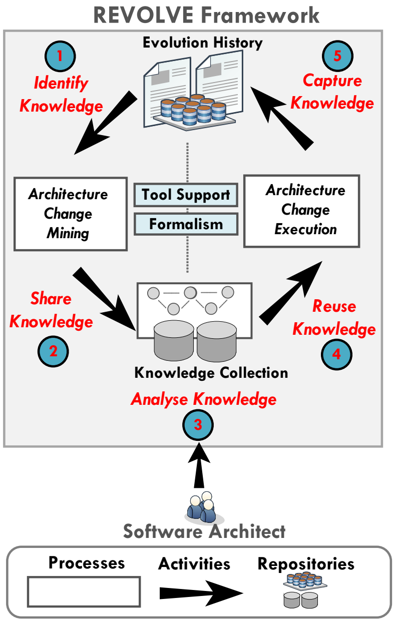


Figure 3.2: REVOLVE - Integrated Views of Architecture Change Mining and Change Execution.

Process	Activity	Repository	Evidences
Architecture Change Mining	*Identify Reuse Knowledge*	Evolution History	[S7, S9, S10, S17]
	Share Evolution Knowledge	Knowledge Collection	[S8, S9, S10, S17]
	Analyse Reuse Knowledge	Knowledge Collection	[S9, S10, S17]
Architecture Change Execution	*Reuse Evolution Knowledge*	Knowledge Collection	[S1, S2, S3, S4, S5, S6, S8, S11, S12, S13, S14, S15, S16, S18, S19, S20, S21, S22, S23, S24, S25, S26, S27, S28, S29, S30]
	Capture Reuse Knowledge	Evolution History	[S7]

Table 3.3: Processes, Activities and Repositories of Framework to Classify Reviewed Studies.

3.4 Results Categorisation and Reuse Knowledge Taxonomy

Our discussion of results uses the classification framework for reuse knowledge from Section 3.3.3. Central to the REVOLVE framework are a set of processes, activities and repositories (in Table 3.3. The processes encompass architecture change mining as a complementary and integrated phase to change execution - a concept partially realised in only one of the reviewed studies [S7]. We also present the relative distribution of the five activities of the REVOLVE framework.

Please note that some of the studies cover different activities of the REVOLVE framework. For example studies [S9, S10, S17, S13] both represent research on *identifying* and *sharing reuse knowledge*. Similarly the study [S7] represent *capturing* and *identifying* reuse knowledge. Table 3.3 summarises the involved processes, their corresponding activities and associated repositories as well as identified studies as concrete evidence of the claims. In Figure 3.2, it is vital to highlight the complementary role of tool support and formalism in ACSE. For example in Figure 3.2, to support automation of the activity for reuse knowledge identification the solution must provide a tool or a prototype to analyse architecture evolution histories that contain evolutionary data of significant size and complexity. A lack of tool support increases the complexity of architecture evolution process, process scalability (changes from small to large systems), error proneness in change implementation.

3.4.1 A Taxonomical Classification of Evolution Reuse Knowledge

The taxonomy defines a systematic identification, naming and organisation of reuse approaches into groups which share, overlap or are distinguished by various attributes. A taxonomical classification provides an insight into the commonality or distinction of research themes as denoted in Figure 3.3. We explicitly discuss two distinct classification types of evolution reuse research as generic and thematic classification of literature. A solution-specific classification will be introduced in Sections 3.5 and 3.6.

1. *Generic Classification* is derived based on a review of studies and the guidelines from [Medvidovic 2000] that helped us to refine classification attributes based on studies for analysing the role of reuse knowledge in architecture evolution. In Figure 3.3, the literature suggests the role of reuse knowledge in ACSE is classified into *methods and techniques that enable change reuse in ACSE* (26 studies, i.e., 87% approx) and *empirical discovery* (4 studies, i.e., 13% approx) *of reuse knowledge and expertise* by exploiting evolution histories.

2. *Thematic Classification* provides details about the predominant research themes based on time and type of evolution. In the following, we focus on a taxonomy of identified research themes based on the classification in Figure 3.3.

- **Evolution Styles** [S1, S5, S8, S11, S13, S21, S23] are inspired by a conventional concept of architecture styles that represent a reusable vocabulary of architectural elements (component or connectors) and a set of constraints on them to express a style [Pahl 2009]. Evolution styles focus on defining, classifying, representing and reusing frequent evolution plans [S1, S11] and architecture change expertise [S5, S8, S13, S21]. *Style-based* approaches represent 22% of the reviewed studies addressing *corrective* and *perfective* changes implemented as design-time evolution. In the style-driven approaches, we observed a trend towards structural evolution-

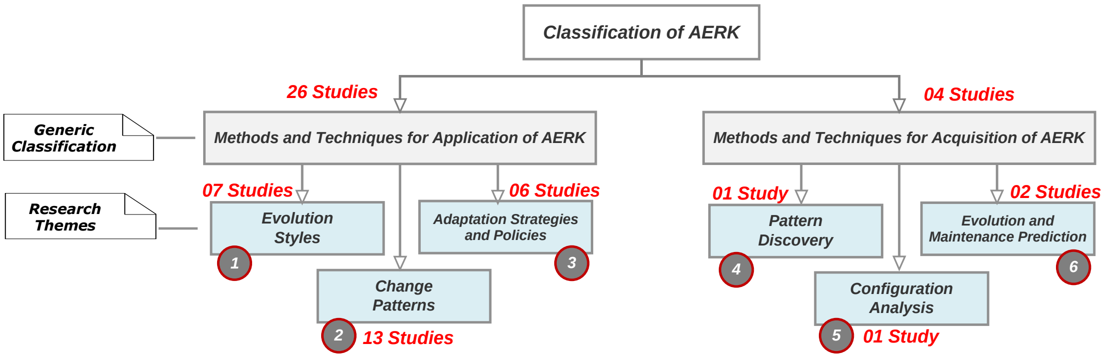


Figure 3.3: A Taxonomical Classification of Architecture Evolution-Reuse Knowledge.

off-the-shelf [S13, S21] and evolution planning [S1, S8] with time, cost and risk analysis to derive evolution plans.

- **Change Patterns** [S2, S6, S12, S14, S15, S16, S17, S20, S21, S24, S27, S29] exploit the same idea as design patterns [Gamma 2001] that aims at providing a generic, repeatable solution to recurring design problems. In contrast, change patterns follow reuse-driven methods and techniques to offer a generic solution to frequent evolution problems. Pattern-based solutions focused on *corrective*, *adaptive* and *perfective* changes supporting both *design-time* as well as *run-time* evolution. *Adaptation* and *reconfiguration patterns* [S16, S19] are the run-time evolution solutions. The solutions also address the *co-evolution of processes* [S29], *requirements* [S2] and underlying *architecture models*. In addition, a number of studies proposed *language-based formalism* [S6, S12, S14, S15] to enable reuse in architectural migration and integration. Unlike styles that only use model-driven evolution, pattern-based changes are expressed as different techniques using model transformations [S2, S29], state transitions [S16, S19] and change operationalisation [S27].

- **Adaptation Strategies and Policies** [S3, S4, S25, S26, S28, S30] focus on reuse and customisation of *adaptation policies* [S3, S4], reusable and *knowledge-driven strategies* [S25, S26, S30] and *aspects* [S28] to support the reuse of policies in self-adaptive architectures. With a recent emphasis on autonomic computing, and growing demand for highly available architectures, reuse-driven strategies aim to provide knowledge-driven reuse at run-time. Run-time reconfigurations of architectures are also highlighted in the MAPE-K reference model [Ganek 2003].

- **Pattern Discovery** [S19] represent methods and techniques for post-mortem analysis of evolution history (version control [S19] systems) to discover recurring changes as pattern instances. Pattern-based knowledge discovery mechanisms is presented in a single study of this review.

- **Evolution and Maintenance Prediction** [S9, S10] focuses on prediction of maintenance and evolution efforts for software architectures. We included two studies in which [S9] represents a set of change scenarios for predicting perfective and adaptive maintenance tasks in architectures. In [S10], based on an architectural evaluation and maintenance prediction, the required maintenance and evolution effort for a software system can be estimated [S10].

- **Architecture Configuration Analysis** [S7] exploits configuration management techniques to analyse architectural configurations [S7]. It focuses on mining architecture revision histories to capture *evolution* and *variability* in order to represent cross-cutting relationships among evolving architecture elements.

3.4.2 Definition of Architecture Evolution Reuse Knowledge

The systematic review question SR-Q1 addresses how architecture evolution reuse knowledge is defined and expressed in the context of ACSE and is answered in this section. After we have defined architecture evolution reuse knowledge here, we answer SR-Q2 (application of reuse knowledge in Section 3.5) and SR-Q3 (acquisition of reuse knowledge in Section 3.6).

In the reviewed studies, we observed that interpreting and assessing individual studies as isolated solutions to a specific research problem lacks consistency in representing what exactly defines reuse knowledge and how it is classified and expressed in literature. More specifically, the taxonomical classification (cf. Section 3.4.1) suggests a lack of consensus and definition for AERK is primarily due to a) different types and time constraints of architectural evolution and b) solution specific interpretation of the evolution reuse. We discuss both of these below.

Architecture Evolution

In the reviewed literature, architecture evolution refers to design-time changes [S1, S2, S6, S13, S20] or run-time reconfigurations [S3, S4, S16, S25] as perfection, adaptation or corrections in architectural structure and behaviour [Williams 2010]. While analysing the titles, keywords and abstracts of included studies, we observed that the term *evolution* (also including evolving, evolve, co-evolution) has six variations as *change* (also including changing), *Reconfiguration*, *Adaptation*, *Restructuring*, *Update*, *Transformation* and *Migration*. The reasons for distinctive terminologies are:

- Types of Architecture Changes as *Corrective*, *Adaptive* (also *Reconfigurative* [S16, S19]), *Perfective* (also *Updative* [S23], *Restructurive* [S21], *Transformative* [S5], *Migrative* [S6]). With a more conventional interpretation of ISO ISO/IEC 14764 and architectural change characterisation in [Williams 2010], we did not find any study to support *preventive* changes. This indicates that existing work lacks support for reuse in pre-emptive and pro-active evolution of architectures [Mens 1999].

- Time Constraints of Changes as highlighted in Figure 3.4 refers to *Evolution*, *Change*, *Update* and *Restructure* for design-time evolution [Medvidovic 1999], while *Reconfiguration* and *Adaptation* refer to run-time evolution [Ganek 2003]. In Figure 3.4, there is a clear inclination towards style-driven approaches, evolutionary plans and model co-evolution for design-time (a.k.a. static evolution). In contrast, run-time (a.k.a. dynamic evolution) is focused on self-adaptation and runtime reconfigurations reflected by studies published in 2004 and 2009.

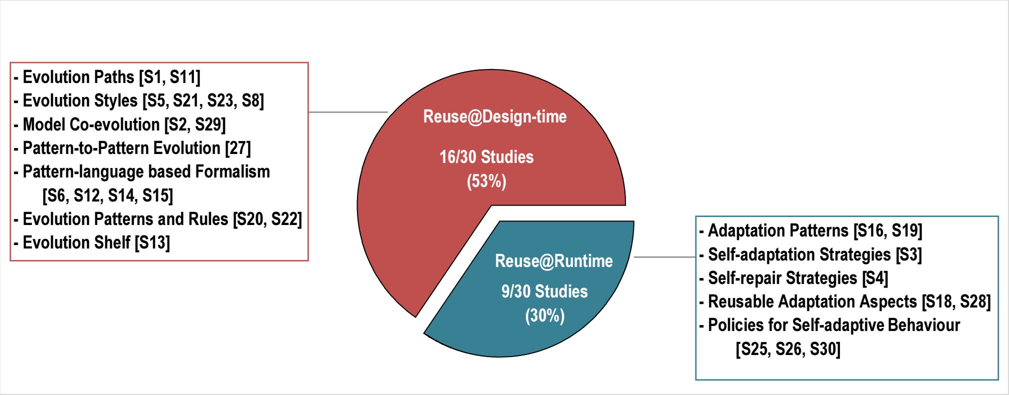


Figure 3.4: Study Percentage Distribution - Time Constraints of Evolution.

This suggests that evolution is an unclear term in the context of types and time of architectural changes, thus making it hard to implicitly derive a unified or aggregated definition for evolution reuse knowledge. Due to a characterisation of architectural change types [Williams 2010] and times of evolution [Buckley 2005], a clear consensus or unified definition is not possible. In fact, it would only limit the acceptance of the concept with a narrow view based on available evidence. However, an aggregated definition of evolution reuse knowledge is important to classify and compare the existing research, see Figure 3.3.

Architecture Evolution Reuse

In the reviewed studies, evolution reuse is expressed as evolution styles, change patterns, and adaptation strategies and policies in Figure 3.3. An interesting observation is that although novel as methodical approaches, both evolution styles and change patterns conceptually extend the more conventional concepts of architecture styles [Pahl 2009] and design patterns [Gamma 2001] to represent evolution expertise. *Evolution styles* [S1, S13, S21] primarily aim at defining, classifying, representing and reusing frequent corrective and perfective changes as a design-time activity. In contrast, *change patterns* [S2, S16, S19] promote the 'build-once, use often' philosophy of to offer a generic, repeatable solution to frequent adaptive, corrective and perfective changes as design-time and run-time-time evolution. The concept of reusable adaptation strategies and policies is only represented in the context of reuse plans [S3, S4, S25] and aspects [S28] for self-adaptive architectures.

Once we have identified the relative representation and expression of evolution and reuse, we provide a consolidated view of architecture evolution reuse knowledge in the context of ACSE. We provide an aggregated definition of Architecture Evolution-Reuse Knowledge (AERK) 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.

In the existing literature, the generic and repetitive solutions are predominantly expressed as evolution styles and patterns. In addition, frequent evolution operations represent addition, removal or modification of architecture elements as design-time change or runtime adaptation. Some studies [S1, S11, S13, S20] implicitly denoted reuse as a first-class abstraction - by operationalising and parametrising changes - to resolve recurring evolution tasks.

3.5 Application of Evolution Reuse Knowledge

Based on the generic and thematic classification in Section 3.4, we now investigate the existing methods and techniques that enable reuse-driven evolution and adaptation in software architectures, i.e., those that apply AERK (SR-Q2). A systematic identification and comparison is particularly beneficial to gain an insight into aspects of *problem-solution mapping*, *architecture evolution characterisation*, or to *assess formalisms and tool support*. The comparative analysis is presented as a number of structured tables (Table 3.4, Table3.5). In this section, a thematic coding process has been employed to identify the comparison attributes (cf. Table 3.2) and to provide an answer to the SR-Q2, i.e., *What are the existing methodologies and techniques that support application of reuse knowledge to evolve software architectures?* (in Section 3.5.1). We also compare the methodologies and techniques to analyse a collective impact of existing research that enhance evolution reuse (in Section 3.5.2).

3.5.1 Methods and Techniques for Application of Reuse Knowledge

For each of the reviewed study, the problem and solution views are explicitly captured (cf. Table 3.2) and represented as generic and documentation specific items (ID = 5 is *Research Problem* and ID = 6 *Proposed Solution*). We also combine the problem and solution views that are related in Table 3.4.

For example, the studies [S1, S11] address the problems of evolution planning and trade-off analysis by applying reusable evolution strategies. While the comparison view is represented with a set of comparison attributes in Table 3.2. Based on the classification of research themes, we focus on answering SR-Q2 with Table 3.4. It has three columns associated with the following aspects:

- *Problem View* - Why is there a need for reuse knowledge to address recurring evolution problems?
- *Solution View* - How do solutions provide methods and techniques to address these research problems?
- *Comparison View* - What are the trends, type, means and time of evolution, formalism and tool support, architectural description notations and evaluation methods? See Table 3.5 for details.

Note that due to the classification scheme (*styles vs. patterns vs. strategies and policies*), we denote adaptation patterns [S16, S19] as a sub-theme of change patterns [S2, S17].

3.5.2 Comparison of Methods and Techniques for Evolution Reuse

In order to go beyond an analysis for individual studies, a holistic comparison of existing research based on comparison attributes including their objective and concrete evidence is provided in Table 3.5. We compare available methods and techniques based on comparison attributes CA1 to CA12 (cf. Table 3.2). The comparison of research methodologies for reuse knowledge-driven

Research Problem	Solutions (Methods and Techniques)	Studies
Evolution Styles		
How to enable evolution Planning and trade-off analysis?	**Evolution Paths** - to plan and apply reusable evolution strategies.	[S1, S11]
How to achieve recurring structural evolution of architecture?	**Evolution Shelf** - library of reusable and reliable evolution expertise	[S13]
How to enhance change reusability and architecture consistency?	**Update Styles** - reuse expertise for restructuring and updating architectures.	[S21, S23]
How to exploit architecture knowledge as an asset for architecture evolution?	**AK-driven evolution styles** | - use of AK as evolution styles styles to constrain and trigger evolution	[S8]
How to reuse in transformation and refinement of component-model to service-driven architectures?	**Style-based Transformations** - to achieve migration from components to business-driven service architecture.	[S5]
Change Patterns		
How to Co-evolve process, requirements with architectures?	**Co-Evolving Models** - reusable patterns to enable co-evolution in process and requirements to their underlying architectures.	[S2, S29]
How to enable a continuous runtime adaptation of architectures?	Patterns - reuse @ runtime to support architectural reconfigurations and self-adaptations.	[S16, S19]
How to exploit the reuse of design methods, documents and process for architecture migration and evolution ?	**Pattern-to-Pattern Evolution and Integration** - evolution operators and design documents to tackle requirement and architecture changes [S27]. Model-based migration and integration of process-centric architecture models [S12, S15].	[S27, S12, S15]
How to enable an incremental migration of legacy architecture by means of reusable decision models?	**Pattern Language-based Formalism** - to facilitate a piecemeal migration of architecture models.	[S6, S14]
How to effectively manage evolution at different architectural abstractions?	**Evolution Patterns and Rules** - to model, analyse and execute architectural transformations at different abstraction levels.	[S20, S22]
Adaptation Strategies and Policies		
How to provide mechanisms for architecture to adapt at run time to accommodate varying resources, system errors, and changing requirements requirements?	**Strategies for Self-adaptation** - supported with stylised architectural design models for automatically monitoring system behaviour falling outside of acceptable ranges, then a high-level repair strategy is selected.	[S3, S4]
How to utilise reusable aspects to develop self-adaptive architectures?	**Reusable Adaptation Aspects** - to reusable aspects and policies to develop self-adaptive architectures.	[S28]
How to efficiently construct system global adaptation behaviour according to the dynamic adaptation requirements?	**Composable Adaptation Planning** - that provides a systematic coordination mechanism to achieve effective and correct composition. It also allows prototyping, testing, evaluation and injection of new adaptation behaviours for component-based adaptable architectures.	[S18]
How to specifying and enact architectural adaptation policies that drive self-adaptive behaviour?	**Knowledge-Based Adaptation Management** - for reasoning and decision-making about the timing and nature of specific adaptations grounded on knowledge-based adaptation policies.	[S25, [S26, S30]

Table 3.4: Methods and Techniques to Enable Reuse knowledge for Evolution and Adaptation.

evolution is based on eight distinct comparison attributes CA1-CA6, CA11, CA12 from the full list (the remaining ones will be covered in the next section). **CA 1:** *What are the identified research trends for reuse in architecture-based evolution and adaptation?*

Objective: The aim is to identify available solutions that support reuse-driven knowledge for ACSE. In addition, an overview of research builds the foundation for a comparative analysis of individual methodologies as discussed below and mapped out later in Figure 3.5.

1. *Evolution-off-the-Shelf* - we observed a trend following evolution styles for structural evolution [S1, S11, S13] in component-based architectures and *evolution planning* [S1, S11] based on time, cost and risk of changes to define alternative evolution strategies. An interesting observation is a recent emergence of *evolution styles* [S8] that exploit architecture knowledge

as an asset to drive evolution-off-the-shelf [S13]. In Figure 3.5, our comparison suggests that evolution style-based approaches only focus on corrective and perfective type changes [Williams 2010]. We could not find any evidence to support adaptive or preventive type evolution.

2. *Pattern and Language-based Formalisms* - pattern-based solutions address the co-evolution of *business processes* [S29] and *requirements* [S2] along with their underlying architecture models. Adaptation [S19] and reconfiguration patterns [S16] support dynamic adaptation as well. Pattern language-based solutions aims at building a system-of-patterns to support *migration* [S6], *integration* [S12, S15] and *evolution* [S14] of component-based architectures. Based on the comparison map in Figure 3.5, we can conclude that pattern-based techniques enable corrective, adaptive and perfective type changes, but do not address preventive change.

1. *Reuse Knowledge for Self-adaptation and Self-repair* - in particular self-adaptive and self-repair techniques reflect the recent emphasis on autonomic computing and growing demands for high-availability architectures. Reuse-driven self-adaptation enables dynamic evolution reflected as reusable *adaptation strategies* for adaptive architectures [S3, S25]. In addition, knowledge-based *adaptation policies* [S4, S26, S30] enhance self-organisation and repair of dynamic adaptive architectures. Self-adaptation strategies are the key to supporting dynamic and high-availability architectures. Unlike styles and patterns, reusable adaptation strategies focus on run-time reuse of adaptation expertise. Moreover, self-repair [S4, S26] policies promise to tackle preventive type changes.

CA 2: *What types of architectural changes are supported to achieve evolution reuse*?

Objectives: The aim is to investigate the type of change support offered by existing ACSE solutions: corrective, perfective, adaptive and preventive changes. This change typology is based on the ISO/IEC 14764 standard and architecture change characterisation in [Williams 2010].

Style-driven approaches focus on corrective and perfective changes (also reported as *updative* [S23], *restructurive* [S21], *transformative* [S5] and *migrative* [S6]). Pattern-based solutions support corrective [S27], perfective [S12, S15, S6, S14] and adaptive change support (also called reconfigurative [S16, S19]). Adaptation strategies and policies, as the name indicates; primarily focus on runtime adaptive [S3, S4, S28] changes.

– represents an attribute not discussed in the reviewed study

++ represents an implicit discussion of the attribute, the remaining is all explicit in literature

Application of AERK	**Research Trends** CA1	**Type of Change** CA2	**Time of Change** CA3	**Means of Change** CA4	**Evolution Formalism** CA5	**Architecture Description** CA6	**Tool Support** CA11	**Evaluation Method** CA12
Evolution Style								
Evolution Paths [S1, S11]	Evolution Plans	Corrective, Perfective	Design-time	Change Operations, Model Transformation++	QVT-based Model Evolution	Acme ADL, UML 2.0++	AEvol	Case Study
Evolution Shelf [S13]	Evolution Styles	Corrective++ , Perfective	Design-time	Model Transformation	QVT-based Model Evolution++	Acme ADL, UML 2.0++	–	Case Study
Update Styles [S21, S23]	Updating Styles [S11], Architecture Style [S10]	Corrective++ , Perfective	Design-time	Model Transformation	Graph Transformation Rules++	ADL++, UML 2.0	AGG [S11] USE[S10]	Case Study
AK-driven Evolution Styles [S8]	AKdES	Corrective++ , Perfective	Design-time	Model Transformation	QVT-based Model Evolution	ATRIUM Meta-model	ATRIUM	Case Study
Style-based Transformations [S5]	Style-based evolution and refinement	Corrective++ , Perfective	Design-time	Model Transformation	Graph Transformation Rules	UML Profile for SOA Graph Model	Poseidon, GTXL	Case Study
Change Patterns								
Model Co-evolution [S2, S29]	Requirements [S2], Business Process [S29]	Adaptive, Corrective	Design-time	Model Transformation	–	UML 2.0 [S2], Graph Model [S15]	VIATRA[S2] N/A [S15]	Industrial Validation [S2] Case Study [S2]
Adaptation Patterns [S16, S19]	Adaptation State-machines	Adaptive, Perfective	Run-time	Reconfiguration Operations++	State Transition	XTEAM, xADL, UML 2.0	REPLUSSE [S6], SASSY [S9]	Case Study
Pattern-to-Pattern Evolution	Pattern-based Evolution and Integration	Corrective Perfective++	Design-time		Jackson's Framework [S27]	Context Diagram	–	Case Study
Pattern Language-based Formalism [S6, S12, S15, S14]	Pattern Migration [S6] Integration [S12, S15] Evolution [S14]	Corrective Perfective Adaptive	Design-time	Change Operations++	– [6], Model-driven Development [S12, S15] RADM [S14]	IDL [6], UML 2.0, XMI [S12, S15] – [S14]	– [6], MDSD Tool Chain [S12, S15] ArchPad [S14]	Migration of Archival System [S6], Case Study [S14]
Evolution Patterns and Rules [S20, S22]	SAEV [S20], TranSAT [S22]	Corrective Perfective	Design-time	– [S6, S14] Model Transformation [S12, S15]	SAEV, ECA [S20] AOSD [S22]	ADL [20] AgroUML [S22]	– [S20] SafArchie [S22]	Case Study
Adaptation Strategies and Policies								
Strategies for Self-adaptation and and Self-repair [S3, S4]	Rainbow Framework [S3], Style-based Adaptation [S4]	Adaptive, Perfective Corrective++	Run-time	Adaptation operators [S3] Repair Strategies [S4]	–	ADL [S3]++ ACME [S4]	Rainbow, Stitch Language [S3]	Case study
Reusable and Composable Adaptation Aspects [S28, S18]	Aspect-orinted Architecture [S28] Composable Adaptation Palnning [S18]	Adaptive, Corrective++	Run-time	Aspect generation and weaving [S28]++ Composable Adaptation Plans [S18]	CaesarJ AO-Programming Language [S28], – [S18]	– [S28], Component Architecture Model [S18]	–	Case study
Adaptation Policies for Self-adaptive Behaviour [S25, S26, S30]	Knowledge-based Adaptation Management	Adaptive, Corrective++	Run-time	Knowledge-based Adaptation Policies	Architectural Adaptation Manager	xADL	KBAAM	Case study

Table 3.5: A Comparison Summary for Research State-of-the-Art on Application of AERK.

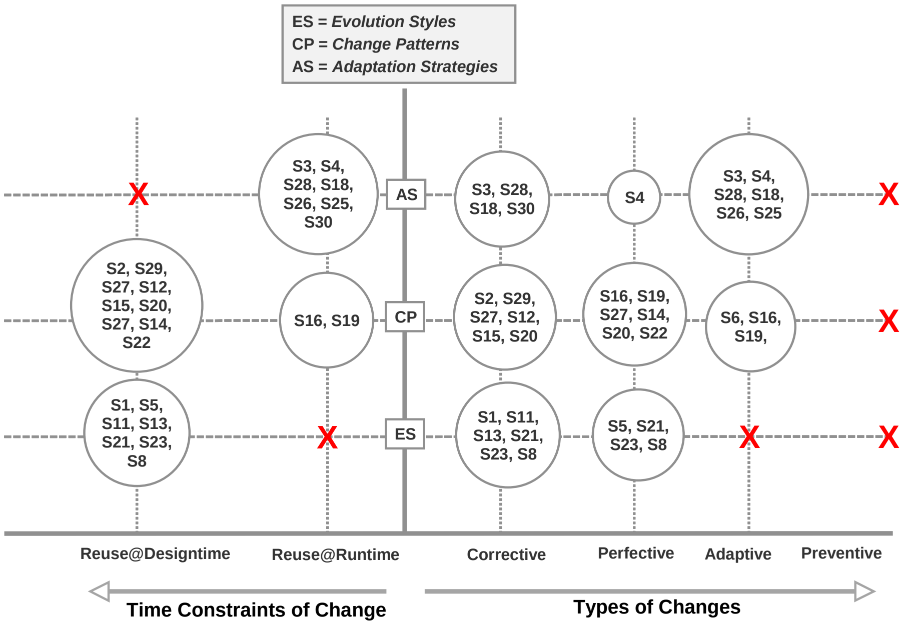

ES = Evolution Styles
CP = Change Patterns
AS = Adaptation Strategies
X
S3, S4, S28, S18, S26, S25, S30
AS
S3, S28, S18, S30
S4
S3, S4, S28, S18, S26, S25
X
S2, S29, S27, S12, S15, S20, S27, S14, S22
S16, S19
CP
S2, S29, S27, S12, S15, S20
S16, S19, S27, S14, S20, S22
S6, S16, S19,
X
S1, S5, S11, S13, S21, S23, S8
X
ES
S1, S11, S13, S21, S23, S8
S5, S21, S23, S8
X
X
Reuse@Designtime
Reuse@Runtime
Corrective
Perfective
Adaptive
Preventive
Time Constraints of Change
Types of Changes

Figure 3.5: A Comparison Map of Research Trends - based on Time and Types of Changes.

CA 3: *How do time aspects affect change implementation during architecture evolution*?

Objectives: The aim is to analyse the temporal aspects [Buckley 2005] in terms of the time (or stage) associated to architecture evolution. The existing evidence suggests:

- **Reuse@Runtime** enables application of reuse knowledge at run-time to achieve dynamic adaptation. Reconfiguration patterns reflect reusable strategies as a consequence of growing demands for autonomic and self-adaptive architectures for run-time evolution [S2, S4, S25, S26, S27]. We could not find any evidence of style-based approaches that facilitate runtime reuse.

- **Reuse@Design-time** enables application of reuse knowledge at design-time to achieve evolution. Style-driven approaches [S1, S13, S8] are heavily oriented towards design-time evolution. In contrast, pattern-driven reuse is aimed primarily at design-time changes [S2, S29, S27] but also support run-time reconfigurations [S16, S19].

CA 4: *What are the existing means of architectural change to achieve evolution reuse*?

Objectives: The aim is to study and compare the change implementation mechanisms and to analyse if there exist any recurring themes among them. We only present the predominant means of change as (at least indicated in five or more studies) as individual methods and techniques are already summarised in Table 3.5.

Evolution operators as the most utilised means of change that could be further classified as *change* [S1, S11, S20, S22, S27], *adaptation* [S19] and *reconfiguration* operators [S16]. *Model transformation* enables design-time evolution as discussed in [S1, S13, S21, S23, S5, S2, S29]. Furthermore, *adaptation plans* exploit *repair strategies* and *aspect weaving* mechanism [S4, S18, S26, S28, S30] for run-time adaptation.

CA 5: *What types of formal methodologies are exploited to support ACSE*?

Objectives: The aim is to analyse the extent to which formal techniques facilitate the modelling, analysing and execution of reuse-driven evolution and adaptation. We only present predominant formal methods (at least indicated in three or more studies).

We observed an overwhelming bias towards model-based architecture evolution that is primarily achieved through model transformation with *QVT* [S1, S11, S13] and also *graph-based specifications* [S11, S10, S5, S12, S15]. This observation is also reported in [Jamshidi 2013b]. The only exceptions are adaptation patterns [S16, S19, S12, S27] that exploit *state-transition* and pattern-to-pattern integration using *Jacksons framework* for architecture evolution.

CA 6: *What are the notations used for architectural descriptions in evolving architecture models*?

Objectives: The aim is to identify the modelling notation used to support architecture evolution. We primarily focus on investigating the role of architecture descriptions in enabling and enhancing architecture evolution (at least three studies).

Three commonly used architectural description notation are *UML 2.0* [S11, S13, S23, S2, S19, S12], *Architecture Description Languages* (ADLs) [S11, S13, S21, S16, S20, S3, S25, S26] and *UML Profiles* [S5, S22, S18]. The primary motive to use ADLs or UML is the availability of extensive research literature and tool support to specify architecture models with model-based verification and transformation to support evolution. Most notable ADLs are ACME and xADL.

CA 11: *What is the available tool support to enable or enhance reuse in architectural evolution and adaptation*?

Objectives: The aim is to analyse the role of automation and tool support in enabling the architect to model, analyse and execute reuse-driven ACSE.

Tool support is significant to assist the architects in decision making and automating complex tasks, especially where there is a need to model and choose among alternative evolution paths [S1, S11]. In the reviewed studies, tool support is generally provided in terms of research prototypes. Automation allows an architect to model [S1, S21], analyse and execute generic, reusable strategies for evolution [S2, S1, S21]. However, there is a mandatory user intervention through appropriate parametrisation and customisation of evolution process to accommodate the human perspective before and after evolution [S6, S9, S11, S12]. Some practical issues and lessons learned regarding tool support for architecture evolution reuse has been reported in [Barnes 2013].

CA 12: *What is the context of evaluation methods to validate research hypotheses or results*?

Objectives: The aim is to analyse the context of evaluation, where evaluation context defines the research environment in which the results are evaluated.

The comparative analysis suggest that validation of the proposed solutions or generated results are heavily based on *surveys*, *controlled experimentation* with case studies [S1, S21, S8] or *evaluation in an industrial context* [S2, S6, S14]. It is evident that solutions are heavily oriented towards case-study based evaluation, usually in a lab-experimentation context. The only exceptions are [S2, S6, S14] that focus on co-evolution of requirements and architectures evaluated in industrial settings.

3.6 Acquisition of Architecture Evolution Reuse Knowledge

In order to complement the methods and techniques that support application of reuse knowledge and expertise to guide ACSE (SR-Q2 in Section 3.5), we now investigate the discovery of evolution-centric reuse knowledge to answer SR-Q3, i.e., *What empirical approaches are employed to discover evolution reuse knowledge?* (in Section 3.6.1). We also compare these methods and techniques to analyse the impact of research for reuse knowledge discovery (in Section 3.6.2).

3.6.1 Methods and Techniques for Acquisition of Reuse Knowledge

We identified *change pattern discovery* [S17], *evolution and maintenance prediction* [S9, S10] and *architecture configuration analysis* [S7] as the three existing means to discover knowledge.

- *Change Pattern Discovery* techniques focus on investigating evolution histories for an experimental identification of recurring change sequences as potential change patterns.

- *Evolution and Maintenance Prediction* methods focus on *maintenance profiles* [S9] and *scenario-based* [S10] prediction of maintenance efforts to enhance or enable architecture evolution.

- *Architecture Configuration Analysis* is centred on an architectural system model that tightly integrates architectural concepts with concepts from configuration management [S7].

These solutions primarily focus on the post-mortem analysis of architecture evolution histories to discover evolution reuse knowledge. In Table 3.6, we summarise the problem-solution mapping to highlight research on knowledge discovery. We can observe a relative lack of focus on establishing and exploiting experimental foundation for a continuous and incremental acquisition of reuse knowledge.

Research Problem	Solutions - Knowledge Discovery Techniques	Studies
	Change Pattern Discovery	
How to empirically discover reusable change operators and patterns	**Evolution History Analysis** - post-mortem analysis of architecture evolution logs and version histories to identify change patterns.	[S17]
	Maintenance and Evolution Prediction	
How to predict efforts of architecture maintenance and evolution?	**Maintenance Profiling** - the architecture is evaluated using so-called scenario scripting and the expected maintenance effort for each change scenario is evaluated for perfective and adaptive changes.	[S9]
	Scenario-based Change Prediction - of complex changes during initial analysis of existing architecture, and how and to what extent the process to elicit and assess the impact of such changes might be improved.	[S10]
	Configuration Analysis	
How to capture and relate changes for architecture configurations?	**Revision History Mining** - captures evolution and variability to represent cross-cutting relationships among evolving architecture elements.	[S7]

Table 3.6: A Summary of Methods and Techniques to Support Reuse knowledge Discovery.

We have identified only a relatively limited number of studies (4/30 of included studies, i.e., 13% approximately), not allowing us any stronger judgments. However, we believe that highlighting the existing literature based on a problem-solution mapping helps us to analyse the existing research and possible future directions as detailed in Table 3.6. In addition, the summarised results in Table 3.6 allow us to assess methodologies for a collective impact of existing research on discovery of evolution-centric reuse knowledge.

3.6.2 Comparison of Methods and Techniques for Acquisition of Reuse Knowledge

We provide the comparison of existing techniques in Table 3.7 that enable reuse knowledge discovery based on six comparison attributes CA7 - CA12 from Table 3.2. The comparative analysis highlights the sources of knowledge, the adoption of empirical approaches and the role of *formalisms* and *tool support*, *type of knowledge discovery* along with *evaluation methods*.

	Knowledge Source CA7	**Type of Analysis CA8**	**Type of Formalism CA9**	**Time of Discovery CA10**	**Tool Support CA11**	**Evaluation Method CA12**
Pattern Discovery	Version Control [17]	Architecture Snapshots	Version Snapshots	Design Time	HEAT	Case Study
Evolution and Maintenance Prediction	Maintenance Profiles [S9]	Change Scenario Evaluation of Evolution	N/A		N/A	
	Change Scenarios [S10]					
Configuration Analysis	Revision Histories [S7]	Configuration Management	N/A		Mae	

Table 3.7: A Summary of Comparison for Research State-of-the-Art on Acquisition of AERK.

We now describe the comparison attributes in detail including their objective and concrete evidence as comparison options used in the cells of Table 3.7.

CA 7: *What types of knowledge sources are investigated to discover evolution reuse knowledge*?

Objective: In order to discover reuse knowledge, existing knowledge sources need to be considered. A source knowledge repository maintains historical ACSE data for knowledge discovery.

- **Pattern Discovery Techniques** exploit *version controls* [S17]) as centrally managed repositories of evolution history. Version controls contain fine-grained traces of evolution data-sets that can be queried and searched to analyse architecture-centric evolution history overtime.

- **Evolution and Maintenance Prediction** utilise maintenance profiles [S9] that represent a set of change scenarios for perfective and adaptive maintenance tasks. More specifically, by exploiting maintenance profile, the architecture is evaluated using so-called scenario scripting and the expected maintenance effort for each change scenario is assessed. Based on architectural evaluation and maintenance prediction, the required maintenance and evolution effort for a software system can be estimated.

- **Architecture Configuration Analysis** investigates architecture *revision histories* [S7]. Revision histories contain datasets for architectural configuration analysis, reflecting evolution and variability of architectures. These are necessary to represent cross-cutting relationships among evolving architectural elements [S7].

CA 8: *What types of analyses are performed on knowledge sources to identify reuse knowledge*?

Objective: to analyse the application of knowledge-discovery mechanisms on knowledge sources. In the context of architecture evolution prediction, *version control snapshots* [S17] techniques are employed to discover change patterns.

CA 9: *What type of formal methods and techniques are utilised for knowledge discovery*?

Objective: is to identify the types of formal methods used for knowledge discovery.

Snapshots of architecture versions are used to discover patterns and possible drifts in architecture from one version to another [S17].

CA 10: *Is knowledge discovered at design-time or run-time*?

Objective: is to distinguish between the techniques for run-time and/or design-time discovery reuse knowledge.

In all of the reviewed studies, evolution reuse knowledge discovery is performed as a design-time activity. We did not find any evidence that highlights maintaining and analysing traces of run-time architectural adaptations.

CA 11: *How are knowledge discovery techniques evaluated*?

Objective: is to compare the type of evaluation methodologies used to validate the knowledge discovery techniques.

The evaluation of knowledge acquisition techniques are primarily based on surveys, controlled experimentation with case studies or evaluation in an industrial context. Existing solutions mainly use case study-based evaluation, usually in a lab-experimentation context.

CA12: *What is the tool support for analysing and discovering reuse knowledge from evolution knowledge sources*?

Objective: is to investigate the extent to which the existing research supports automation and customisation of the knowledge discovery process with support by prototypes and tools.

Tool support is critical, especially where the amount of data or the complexity of the knowledge source is substantial. It is difficult, time consuming and error prone to perform analyses manually.

3.7 Implications of Systematic Literature Review

In this chapter, we presented the results of a systematic review to analyse the collective coverage and impact of existing research that enable or enhance architecture evolution with reuse knowledge. We classified the existing work (Section 3.4) and provided a comparative analysis for methods and technique enabling application (Section 3.5) and acquisition (Section 3.6) of reuse knowledge to guide architecture evolution. We now present a summary of research progress and principle findings to highlight trends and possible future research - also formulating our solution in Chapter 4. A yearly distribution of reviewed studies (research progression to-date) and associated research trends are presented in Figure 3.6. The year 1999 was chosen as the preliminary search found no earlier results related to any of the research questions.

3.7.1 Research Trends and Future Directions

In the context of software evolution, research on reuse-driven architecture evolution is continuously growing over more than a decade (as observed in the reviewed studies from 1999 to 2012). As indicated in Figure 3.6, we did not set a lower boundary for the year of publication in the search process, yet the time-frame of identified studies reflects also the timeframe of emergence

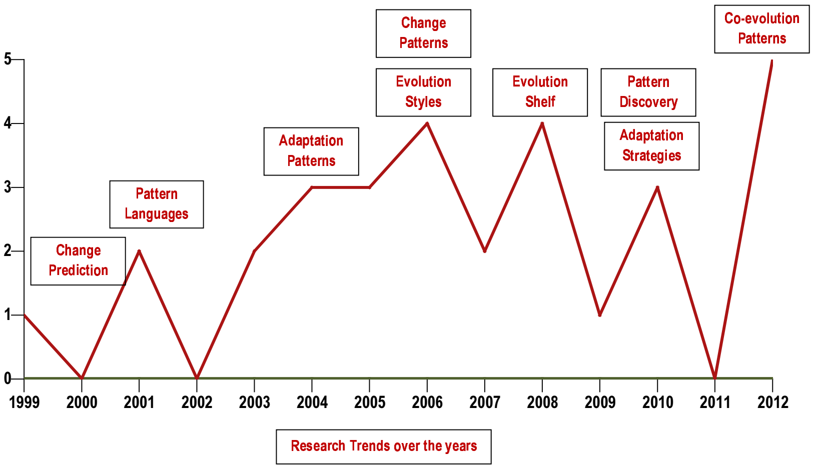


Figure 3.6: Temporal distribution of the primary studies (1999 - 2012).

and maturation of solutions. The trend curve starts in 1999 with a study on predicting architecture maintenance and evolution [S9]. Since 2004, an interesting observation (cf. Table 3.4) is a continuous exploitation of the concept 'evolution styles' to support planning [S1, S11], operationalising [S21] and fostering [S13] of reuse knowledge.

A reflection on research trends and possible future directions is presented in Table 3.8 and Table 3.9 along the aspects of *methods and techniques to enable reuse-driven evolution and discovery of reuse knowledge and expertise*.

Classification	**Methods and Techniques for to Apply AERK**		
Solutions	*Evolution Styles*	*Change Patterns*	*Adaptation Strategies*
Identified Research Trends	Evolution Planning [S1, S11, S8]	Model Co-evolution [S2, S29]	Self-Adaptation and Repair [S3, S4]
	Evolution Paths [S21, S23]	Adaptation Patterns [S16, S19]	Composable Adaptations [S18, S28]
	Evolution Shelf [S13, S21]	Pattern Languages [S6, S12, S15]	Adaptation Knowledge [S25, S26, S30]
Potential for Future Dimensions	**Reuse@Designtime**		**Reuse@Runtime**
	Knowledge-driven Migration Integration and Evolution Model Co-evolution		Reconfiguration Patterns Adaptation Plans and Reusable Infrastructure

Table 3.8: Methods and Techniques for Reuse Knowledge Application.

Research Trends in Reuse Knowledge Application

The identified research themes to express reuse knowledge in architecture evolution are primarily classified as *evolution styles*, *change patterns* and *adaptation strategies*. Evolution styles [S1] are primarily focused on deriving generic evolution plans [S11, S8, S21] to support design-time evolution of architectures. In contrast, adaptation strategies [S3] aim to support reusable adaptation strategies [S18, S28] to support runtime evolution. Only change patterns [S2, S16] could support both design-time and run-time evolution in architectures. More specifically, pattern languages [S6, S12] and architecture co-evolution [S2, S29] are the most notable trends for enabling pattern-driven reusable evolution. Although we only identified 2 studies, adaptation patterns promote reuse in runtime evolution [S16, S19].

Future Research Dimensions for Reuse Knowledge Application

We can identify the need for future research based on time aspects of evolution reuse that include:

- *Reuse@run-time* refers to application of reuse to support reuse-driven dynamic adaptation in software architectures (a.k.a. on-line evolution). In an architectural context for high availability, there is an obvious need to capitalise on generic and off-the-shelf expertise to support reuse-driven self-adaptation [S3, S4, S18, S25]. The IBM autonomic framework [Ganek 2003] - *Monitor-Analyse-Plan-Execute* (MAPE) loop - embodies the *topology*, *policy* and *problem determination knowledge* to derive configuration plans and to enforce adaptation policies to monitor and execute software adaptations. In contrast to studies [S4, S25, S30], we argue that augmenting the conventional MAPE loop with explicit change reuse knowledge can systematically address frequent adaptation tasks. The existing solutions either allow customisation of reusable infrastructure [S3], self-repair [S4] or adaptation aspects [S28] to existing software. However, they lack support for evolution reuse to guide dynamic adaptations. We conclude that when addressing recurring evolution, the potential lies with *fostering* and *reusing* off-the-shelf dynamic adaptations to enable evolution reuse at runtime.

- *Reuse@design-time* refers to application of reuse to support generic and reusable evolution in software architectures (a.k.a. off-line evolution). Existing research clearly focuses on styles and patterns for the reuse of generic evolution plans, change operationalisation and model-based architecture co-evolution. With the REVOLVE framework, our review suggests the need to augment styles [S1, S11, S13, S21] and pattern-driven solutions [S2, S29] with repository mining techniques [S17] to discover reusable evolution strategies.

Research Trends in Reuse Knowledge Acquisition

In contrast to reuse knowledge application, we can observe a clear lack of research on knowledge discovery techniques (only 4 studies) despite an acknowledged need. The primary themes for evolution-centric knowledge discovery represent *pattern discovery, evolution prediction and architecture configuration analysis*. Change pattern discovery aims at investigating version control [S17] systems for post-mortem analysis of evolution histories. Frequent change instances from evolution histories are identified and represented as patterns. Architecture-based prediction of software evolution aims to exploit scenario-based analysis to estimate the efforts of software evolution [S9, S10]. Configuration analysis techniques aim to investigate the evolution-centric dependencies for software architectures [S7].

Classification	**Methods and Techniques for Acquisition of AERK**		
Solutions	*Pattern Discovery*	*Evolution Prediction*	*Configuration Analysis*
Identified Research Trends	N/A	Evolution Scenario Analysis [S10]	Change Configuration Analysis [S7]
	Evolution Paths [S21, S23]	Change Version Mining [17]	Maintenance Profile Analysis [S9]
Potential for Future Dimensions	**Evolution Mining**		
	Analysing Evolution-centric Couplings		
	Evolution Dependency Analysis		

Table 3.9: Methods and Techniques for Reuse Knowledge Acquisition.

Future Research Dimensions for Reuse Knowledge Acquisition

The comparative analysis for knowledge discovery techniques suggest an investigation of evolution-centric dependencies. In particular, we propose *Evolution Mining* that aims at *analysing*, *discovering* and *sharing* explicit knowledge to be *reused* to anticipate and guide architecture change management. In the reviewed studies, there is little evidence of architecture change mining. Our review suggests the needs for empirically derived evolution plans and also the need to analyse evolution dependencies. Such dependency analysis is significant to identify the commutative and dependent changes in order to investigate parallelisation of evolution operations.

The classification framework provides a holistic view of different evolution reuse aspects to be considered in the context of the REVOLVE framework (Figure 3.2). The trends in Table 3.8 and Table 3.9 reiterate the fact that among prominent concerns to tackle are time aspects of evolution. It reflects on the role of formalisms and tool support that can be exploited to leverage conventional data mining techniques for post-mortem analysis of architecture evolution histories. We also identified the needs for a tool chain that could automate the REVOLVE framework with appropriate and minimal user intervention only.

3.8 Summary of Chapter

This chapter as a systematic review complements the concepts introduced in earlier chapters in the context of architecture evolution reuse knowledge (AERK), i.e., knowledge specific to reuse in the evolution of software architecture. Based on a qualitative selection of 30 studies, we investigated the coverage and concerns of reuse knowledge in architecture-centric software evolution (ACSE).

We define what exactly constitutes reuse knowledge in the context of architecture evolution based on the review. We derived a taxonomy that classifies existing and future approaches for reuse-driven evolution that reflects a continuous progression of research over the last decade. The reported results provide us the foundation to develop the solution framework in Chapter 4. Based on the proposed conceptual framework, we distinguish between research efforts on architecture change discovery and mining (4/30 studies, i.e., 13% of the reviewed literature) and architecture change execution (26/30, 87%). A relative lack of focus on empirical identification of reuse knowledge suggests the need of solutions with *architecture change mining* as a complementary and integrated phase for *architecture change execution*.

List of Studies Selected for Systematic Review

- [*S*1] **J. M. Barnes, D. Garlan and B. Schmerl**. *Evolution Styles: Foundations and Models for Software Architecture Evolution*. In Journal of Software and Systems Modeling, 2012.

- [*S*2] **K. Yskout, R. Scandariato, W. Joosen**. *Change Patterns: Co-evolving Requirements and Architecture*. In Journal of Software and Systems Modeling, 2012.

- [*S*3] **D. Garlan, S. Cheng, A. Huang, B. Schmerl, P. Steenkiste**. *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*. In IEEE Computer, 2004.

- [*S*4] **D. Garlan, S.W. Cheng, B. Schmerl**. *Increasing System Dependability through Architecture-Based Self-Repair*. In Architecting Dependable Systems, 2008.

- [*S*5] **L. Baresi, R. Heckel, S. Thöne and D. Varró**. *Style-based Modeling and Refinement of Service-oriented Architectures*. In Journal of Software and Systems Modeling, 2006.

- [*S*6] **M. Goedicke and U. Zdun**. *Piecemeal Legacy Migrating with an Architectural Pattern Language*. In Journal of Software Maintenance: Research and Practice, 2002.

- [*S*7] **A. Hoek, M. Rakic, R. Roshandel and N. Medvidovic**. *Taming Architectural Evolution*. In Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2001.

- [*S*8] **C. E. Cuesta, E. Navarro, D. E. Perry, C. Roda**. *Evolution Styles: Using Architectural Knowledge as an Evolution Driver*. In Journal of Software: Evolution and Process, 2012.

- [*S*9] **P. Bengtsson and Jan Bosch**. *Architecture Level Prediction of Software Maintenance*. In 3rd European Conference on Software Maintenance and Reengineering, 1999.

- [*S*10] **N. Lassing, D. Rijsenbrij, H. Vliet**. *How Well can We Predict Changes at Architecture Design Time*. In Journal of Systems and Software, 2003.

- [*S*11] **D. Garlan, J. M. Barnes, B. Schmerl, 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, 2009.

- [*S*12] **C. Hentrich and U. Zdun**. *Patterns for Process-Oriented Integration in Service-Oriented Architectures*. In 11th European Conference on Pattern Languages of Programs, 2006.

- [*S*13] **O. L. Goaer. D. Tamzalit, M. Oussalah, A. D. Seriai**. *Evolution Shelf: Reusing Evolution Expertise within Component-Based Software Architectures*. In IEEE International Computer Software and Applications Conference, 2008.

- [*S*14] **O. Zimmermann, U. Zdun, T. Gschwind, F. Leymann**. *Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method*. In 7th Working IEEE/IFIP Conference on Software Architecture, 2008.

- [*S*15] **U. Zdun and S. Dustdar**. *Model-Driven and Pattern-Based Integration of Process-Driven SOA Models*. In International Journal Business Process Integration and Management, 2007.

- [*S*16] **H. Gomaa, M. Hussein**. *Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures*. In 4th Working IEEE/IFIP Conference on Software Architecture, 2004.

- [*S*17] **X. Dong, M. W. Godfrey**. *Identifying Architectural Change Patterns in Object-Oriented Systems*. In 16th IEEE International Conference on Program Comprehension, 2008.

- [*S*18] **N. Gui and V. De. Florio**. *Towards Meta-Adaptation Support with Reusable and Composable Adaptation Components*. In IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems, 2012.

- [*S*19] **H. Gomaa, K. Hashimoto, M. Kim, S. Malek, D. A. Menascé**. *Software Adaptation Patterns for Service-oriented Architectures*. In ACM Symposium on Applied Computing, 2010.

- [*S*20] **N. Sadou, D. Tamzalit, M. Oussalah**. *How to Manage Uniformly Software Architecture at Different Abstraction Levels*. In 24th International Conference on Conceptual Modeling, 2005.

- [*S*21] **D. Tamzalit, T. Mens**. *Guiding Architectural Restructuring through Architectural Styles*. In 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, 2010.

- [*S*22] **O. Barais, L. Duchien, A. Le Meu**. *A Framework to Specify Incremental Software Architecture Transformations*. In 31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005.

- [*S*23] **D. Tamzalit, M. Oussalah, O. L. Goaer, A. d. Seriai**. *Updating Software Architectures: A Style-based Approach*. In International Conference on Software Engineering Research and Practice, 2006.

- [*S*24] **Le. Goaer, M. Oussalah, D. Tamzalit**. *Reusing Evolution Practices onto Object-Oriented Designs: An Experiment with Evolution Styles*. In 19th International Conference on Software Engineering and Data Engineering, 2010.

- [*S*25] **J. C. Georgas R. N. Taylor**. *Towards a Knowledge-Based Approach to Architectural Adaptation Management*. In 1st ACM SIGSOFT Workshop on Self-managed Systems, 2004.

- [*S*26] **J. C. Georgas , A. v.d. Hoek , R. N. Taylor**. *Architectural Runtime Configuration Management in Support of Dependable Self-Adaptive Software*. In Workshop on Architecting Dependable Systems, 2005.

- [*S*27] **I. Côté , M. Heisel , I. Wentzlaff**. *Pattern-Based Evolution of Software Architectures*. In European Conference on Software Architecture, 2007.

- [*S*28] **E. Truyen and W. Joosen**. *Towards an Aspect-oriented Architecture for Self-adaptive Frameworks*. In Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2008.

- [*S*29] **P. Jamshidi, C. Pahl**. *Business Process and Software Architecture Model Co-evolution Patterns*. In Workshop on Modeling in Software Engineering, 2012.

- [*S*30] **J. C. Georgas R. N. Taylor**. *An Architectural Style Perspective on Dynamic Robotic Architectures*. In IEEE 2nd International Workshop on Software Development and Integration in Robotics, 2007.

Chapter 4

PatEvol - A Framework for Integration of Architecture Change Mining and Change Execution Processes

Contents

4.1 Chapter Overview

The classification and comparison of architecture evolution reuse knowledge suggests the needs for an incremental process to continuously gather and reuse evolution-centric knowledge (cf. Chapter 3). Moreover, we have organised these research activities as a conceptual model REVOLVE that reflects a theoretical reference to the body of existing research that enables reuse of architecture evolution. In order combine different activities for reuse knowledge discovery and its application we need to provide a concrete framework that comprises of a set of processes and activities to enable acquisition and application of architecture evolution reuse knowledge. REVOLVE is a conceptual reference to the research state-of-the-art on evolution reuse (cf. Chapter 3), while the PatEvol framework in this chapter represents a set of concrete processes and activities to support the acquisition and application of reuse knowledge. In order to develop this framework, we propose that reuse knowledge-driven evolution in software architectures can be achieved by following a two-step process - a solution that is lacking in the existing research.

- *Step 1* - includes a continuous acquisition of reuse knowledge.
- *Step 2* - relies on application of discovered knowledge to facilitate the execution of frequent architecture changes.

Recent research has demonstrated that reuse of recurring adaptation strategies and policies saves about **40%** of the effort for architecture evolution compared to an ad-hoc and once-off implementation of adaptive changes [Cámara 2013]. In this chapter, we present the PatEvol framework that provides an integration of architecture change mining and change execution processes to facilitate evolution reuse. We also explain the underlying processes and activities (framework elements) of the PatEvol framework to enable acquisition and application of reuse knowledge. By process integration, we mean that *change mining process enables a continuous acquisition of evolution-centric knowledge by analysing architecture evolution histories, and then discovered knowledge can then be reused to support architecture change execution*. Knowledge acquisition enables a continuous and incremental discovery of an explicit evolution-centric knowledge from established knowledge sources. Knowledge application refers to utilising the discovered knowledge to enable reuse in architecture evolution as presented in Figure 4.1.

4.2 PatEvol - Pattern-driven Architecture Evolution Framework

We propose that evolution-centric reuse-knowledge acquisition requires a continuous investigation of the sources of evolution knowledge to identify reusable evolution expertise [Gall 1997, Zimmermann 2005]. In order to achieve this, we propose architecture change mining as a complementary and integrated phase to architecture change execution, as illustrated in Figure 4.2 - PatEvol framework. More specifically, change mining as a sub-domain of data mining [Bengtsson 1999, Lassing 2003] and more specifically *software repository mining* [Gîrba 2006, Kagdi 2007] entails the

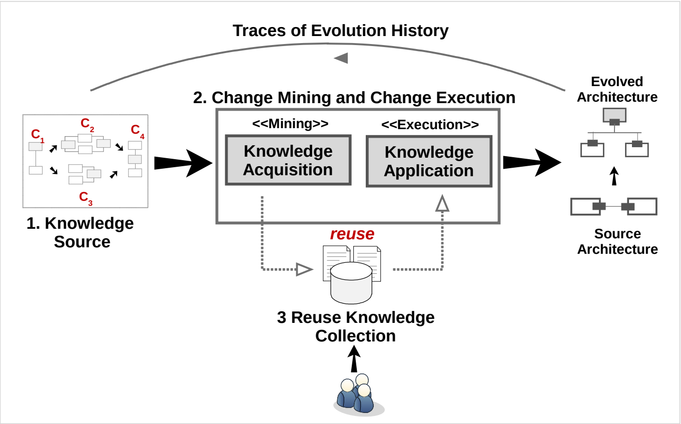


Figure 4.1: Overview of the Architecture Change Mining and Change Execution Processes.

(automated) extraction of hidden and predictive information from large data sets regarded as software evolution histories [Zimmermann 2005, Kagdi 2007]. In contrast, change execution as a subdomain of software evolution [Lehman 2003] and more specifically *architecture-driven change management* [Williams 2010, Medvidovic 1999] refers to a systematic mapping the problem-solution views and the application of the discovered solutions to address recurring evolution problems [Garlan 2009, Le Goaer 2008].

4.2.1 Elements of the PatEvol Framework

In the following, we provide a systematic presentation of the main building blocks of the framework, so called concrete elements of the PatEvol framework. Each of the element is presented along with its role in the framework as summarised in Table 4.1 and illustrated in Figure 4.2. We propose PatEvol, as an overall framework that outlines a set of process and activities to enable discovering and reusing evolution-centric knowledge. The **processes** in the framework define what needs to be done and the **activities** in a process demonstrate how it is done [Fayad 1997].

Framework Processes	Process Activities	Repositories
Architecture Change Mining	Classification of Change Operations and Dependencies	Architecture Change Logs
	Discovery of Architecture Change Patterns	
	Composition of Change Patterns Language	
Architecture Change Execution	Specification of Architecture Changes	Change Patterns Language
	Selection of Architecture Change Patterns	
	Pattern-based Evolution of Architecture	

Table 4.1: Summary of the Processes, Activities and Repositories in the PatEvol Framework.

- **Processes in the Framework**: The processes (indicated as a white square - Figure 4.2) represent two distinct phases of the framework as architecture-centric change mining and architecture change execution processes as outlined in Table 4.1.

- **Activities inside Processes**: Each process comprises of a set of underlying activities (indicated as blue rectangle - Figure 4.2) that highlight the distinct phases for knowledge discovery and its application in evolution. Each of the change mining and change execution processes are comprised of three activities listed in Table 4.1.

- **Role of Repositories in the Framework**: In addition to the core processes and activities, the role of the repositories (a.k.a. knowledge collections) to contain evolution-centric knowledge. More specifically, the knowledge source or architecture change logs [Yu 2009, Wermelinger 2011, ROS-Distributions 2010] represent a central repository that contains fine-grained instances of architecture change and provides a foundation for change mining. We propose a pattern language as a collection of inter-connected change patterns with a formalised vocabulary and grammar [Alexander 1999, Goedicke 2002] with patterns that build on each other to provide a solution to recurring evolution problems.

4.3 Processes and Activities in the PatEvol Framework

In this section, a discussion of the framework processes and activities allow us to highlight the thesis contribution, processes and activities are detailed in dedicated chapters of the thesis.

4.3.1 Process I - Architecture Change Mining

The role of change mining is fundamental in enabling a systematic investigation into the history of sequential architecture changes to analyse recurring change operationalisation that represent the potential change patterns. Our objective of change mining is identical to that of software evolution analysis [Zimmermann 2005, Lassing 2003] that exploits the history of a software system to analyse its present state and to predict its future [Zimmermann 2003]. However, architecture change mining is aimed at employing a set of (automated) techniques for extraction of hidden predictive information in terms of investigating architecture changes instances from change logs as evolution history that have been aggregating over time [Kagdi 2007]. In order to obtain an accurate insight into history of architecture evolution, change mining process relies on the availability of an explicit knowledge source that can be systematically investigated to extract evolution-centric knowledge.

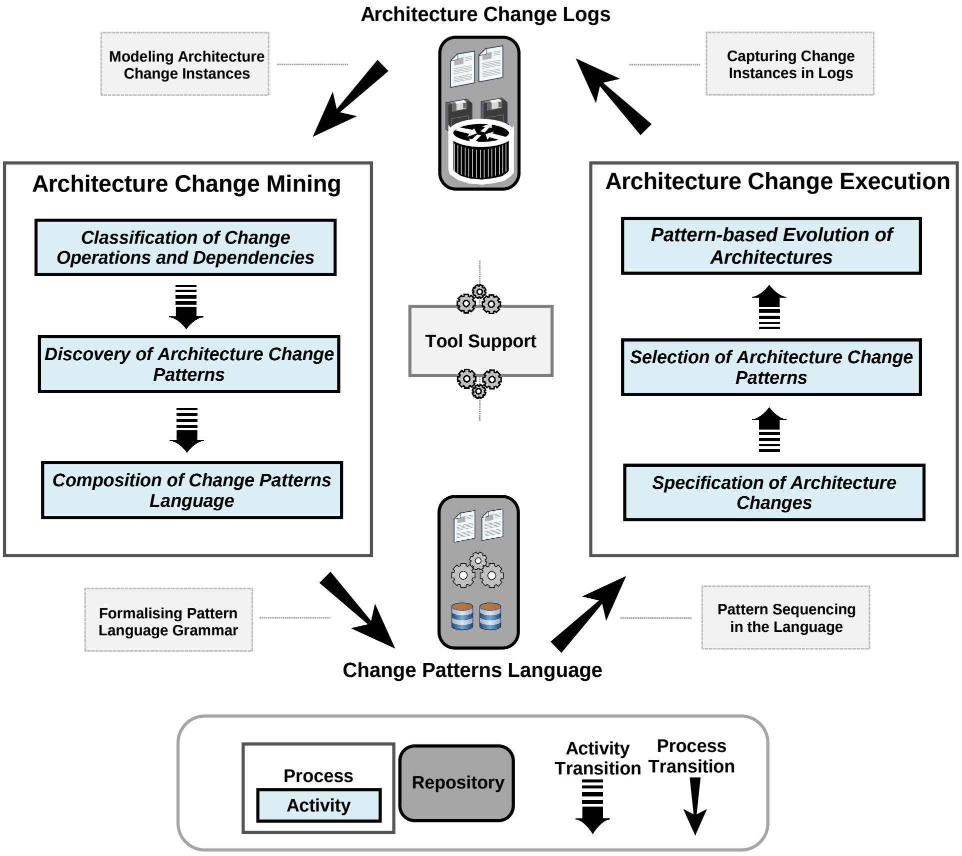

Architecture Change Logs
Modeling Architecture Change Instances
Capturing Change Instances in Logs
Architecture Change Mining
Classification of Change Operations and Dependencies
Discovery of Architecture Change Patterns
Composition of Change Patterns Language
Tool Support
Architecture Change Execution
Pattern-based Evolution of Architectures
Selection of Architecture Change Patterns
Specification of Architecture Changes
Formalising Pattern Language Grammar
Pattern Sequencing in the Language
Change Patterns Language
Process
Activity
Repository
Activity Transition
Process Transition

Figure 4.2: Overview of the PatEvol Framework.

Therefore, we exploit architecture change logs that provide us with fine-grained details about architecture change instances that vary from a simple change like adding a port to a component to a complex change like integrating, replacing or decomposing the components in existing architecture. In a collaborative environment for architectural development and evolution, a change log represents a source of evolution knowledge to facilitate with 'post-mortem' analysis for architectural change instances. A change log in the PatEvol framework consists of individual change instances from architecture evolution case studies [EBPPCaseStudy , 3-in-1 Phone System 1999].

Automation and User Intervention in Change Mining Process - Considering architecture change analysis in [Barnes 2013], in addition to automation; user intervention is also required - human-centric feedback and supervision - for the change mining process. More specifically, during change mining, the complex operational tasks such as change operation analysis and pattern discovery are semi-automated as some user intervention is also required. The user intervention is supported with some pattern discovery parameters that enable the customisation of the discovery process. The scalability of pattern-discovery process is supported with a prototype G-Pride

(Graph-based Pattern Identification) that enable automation and parametrised user intervention of pattern discovery process.

Modelling Architecture Change Instances from Logs

In order to systematically investigate change logs, we need to formalise individual change instances captured in the log that also refers to pre-processing of change logs data for change mining. The need for a formal and structured representation is driven by the fact that raw representation of log data is complex (mainly due to *dependencies* and *sequences* of changes), and therefore its analysis are time consuming and error prone. Based on graph-theoretic details in Chapter 2, we exploit graph-based notation to formalise change instances in the log as a graph [Ehrig 2004]. The nodes and edges of change log graph represent change operations and their sequencing respectively on architecture elements. Graph-based representation of the log data is beneficial for a formal semi-automated and efficient analysis of fine granular change instances in the logs. In addition, modelling architecture changes as a graph, a significant benefit lies in utilising the graph matching and sub-graph mining [Jiang 2012] techniques to investigate change representation and operational dependencies that enables discovery of recurrent change sequences in the log. The goal of this activity is to formalise the change log data that is represented as an architecture change log graph detailed in Chapter 5.

In the following we introduce the activities of the framework that are focused on log-based *classification of architecture change operations and operational dependencies*. The ultimate outcome of the change mining process is *pattern discovery* that provides us with the foundation for the composition of a *change pattern language* for evolution in software architectures.

- **Activity I - Classification of Change Operations and Dependencies -** Once log data is formalised as a graph, a more intuitive approach to gain a systematic insight into architectural changes is to analyse how changes are represented on architecture elements over a period of time. Here a graph-based formalism provides us with an option to exploit graph-matching - comparing change instances - to analyse the operational composition and characterisation of changes [Jiang 2012]. Such an analysis requires details about the composition of architecture changes and the possible operational representation of change instances. The outcome of this activity is a taxonomical classification of change instances as *atomic*, *composite* and *sequential* change operations. In addition, a fine-granular change operational classification is vital to distinguish between *commutative* and *dependent* changes in the log detailed in Chapter 6. Change dependency analysis helps to analyse *the extent to which architectural change operations are dependent or independent of each other* (if architecture change operations could be parallelised).

- **Activity II - Discovery of Architecture Change Patterns -** The outcome of **Activity I** is a taxonomical classification of architecture change operationalisation. The frequency of a change determines if a certain type of change occurs repeatedly over time (as captured in the change logs). This motivates us to exploit change sequence abstraction to determine frequently occurring changes that represent potential *change patterns* discovered from change logs. A change pattern represents a generic and potentially reusable operationalisation that could be a) identified as a recurrent solution, could be b) specified once and c) instantiated multiple times to support potential reuse in architecture evolution [Tamzalit 2010, Yskout 2012]. Again, we aim to exploit graph-based formalism and utilise sub-graph mining [Jiang 2012] as a knowledge discovery technique to discover recurrent change sequences in the log. The intent and the impact of the discovered change patterns are visualised that helps a pattern author to specify them in a pattern template. The outcome of the pattern discovery activity is a collection of discovered patterns from logs that allow us to derive a change pattern language detailed in Chapter 7.

- **Activity III - Composition of Change Patterns Language -** The pattern language is formally composed of a) a classified composition of discovered patterns and their variants (*language Vocabulary*) along with a b) set of rules that govern the relations among pattern elements (*language Grammar*) to create an c) interconnection-of-patterns (*pattern Sequencing* in the language). The proposed pattern language provides a collection of change patterns that support reusable solutions to recurring evolution problems. Reuse-knowledge in the proposed pattern language is expressed as a formalised collection of interconnected-patterns detailed in Chapter 8. Patterns as a generic and solution-specific knowledge to resolve recurring evolution problems could not be invented. Patterns along with their possible variants must be discovered by analysing the problem space and the solution context. We summarise the outcome architecture change mining process as:

  1. Enabling 'post-mortem' analysis of architecture evolution histories to discover operationalisation and patterns that could be reused to guide change management.

  2. Language as a system of pattern allows a mapping of patterns as reusable solutions to

recurring architecture evolution problems. The role of a pattern language is central in promoting patterns to achieve reuse and consistency in evolution of architectures.

4.3.2 Process II - Architecture Change Execution

As a consequence of frequent business and technical change cycles, software systems and ultimately their architecture becomes prone to a continuous maintenance and evolution. This motivates the need to unify the concepts of software repository mining [Gîrba 2006, Zimmermann 2005] and software evolution [Lehman 2003, Mens 2008] in a way that change mining provides discovered knowledge to complement and guide change execution. The research state-of-the-art [Breivold 2012] on ACSE lacks such an integrated approach that exploits architectural change mining to guide architecture change execution process. Such an integrated solution can relieve an architect of routine evolution tasks with reuse to support a systematic change execution whenever the needs for architectural evolution arises [Williams 2010]. The needs for taming architectural changes for their future reuse during evolution is also highlighted in [van der Hoek 2001]. In the context of change execution in Figure 4.2, language provides a reuse knowledge base for pattern-driven architecture evolution. During evolution, change instances are captured in the log for an incremental update of evolution history to establish the loop for knowledge acquisition and knowledge application.

- **Activity I - Specification of Architecture Changes -** Change specification allows representing the changes on a source architecture that leads to its evolution. In this context, a declarative specification enables an architect to represent the context of architectural change that contains the a) source architecture, b) any constraints on the architecture model (to preserve specific architecture elements during evolution) and c) architecture elements that need to be added, removed or modified to achieve architecture evolution. Change specification allows representing the intent and scope of individual changes explicitly in the source architecture model. During change specification an architect may want to specify architectural constraints to protect the specific architectural elements from consequences of change before and after evolution. In order to enable evolution, change specification is the first step to represent a transition of source architecture towards an evolved architecture. We discuss details of architecture change specification in Chapter 8.

- **Activity II - Selection of Architecture Change Patterns -** Once architectural changes are specified, the pattern language provides an interconnected collection of patterns as a problem-solution mapping based on a given context of evolution (from change specification). However, pattern selection is a complex problem [Kampffmeyer 2007] and in order to query the language the user must know at least the structural composition of the language as well as a detailed knowledge about existing patterns in the language. We adopt the design space analysis [MacLean 1991, MacLean 1995] for a systematic pattern selection from language collection with guidelines in [Zdun 2007]. Design space analysis is a methodology to address design-related problems in Human Computer Interaction (HCI). However, it has been applied for pattern selection [Zdun 2007] and provides us with a three step selection process. For example, following design-space analysis, change specification enables querying the language using the Question-Option-Criteria (QOC) methodology [MacLean 1991] to retrieve the appropriate pattern that provides the potential reuse of change operationalisation to enable architectural evolution. More specifically, in QOC *Question* refers to specification of architecture change, *Option* represents the available patterns in a given evolution scenario (provided with change specification), and *Criteria* represents the consequences of applying the given pattern - details provided in Chapter 8.

- **Activity III - Pattern-based Evolution of Architectures -** The retrieved pattern(s) can be applied and they abstract the operational execution details and provide a generic, reusable solution to architectural change execution. We present details of pattern-based architecture evolution in Chapter 8. The outcome of change execution is:

    1. A declarative specification of the change request by enabling selection of appropriate pattern sequences for deriving reusable evolution based on given evolution scenarios.
    2. Pattern language provides a method of systematic reuse based on an incremental application (by selecting and applying a sequence) of patterns from a collection.

4.3.3 Types of Collection in the Framework

In the PatEvol framework, the role of repositories is central as the collection of knowledge in terms of extracting and maintaining reusable operationalisation and patterns during change execution.

Collection I - Change Log as a Source of Architecture-centric Evolution Knowledge

In order to ensure an incremental discovery of evolution reuse-knowledge, it is required to capture and maintain the traces of evolution by means of a transparent and centrally manageable collection of change instances [Kagdi 2007, Zimmermann 2005]. A careful selection is required in terms of utilising and establishing the collection as an active repository infrastructure that facilitates a flexible storage and retrieval of architectural changes. In comparison to the version control systems that focus on capturing (implementation-level) source code changes, change logs trace evolution when changes (at design-level) are applied to software components and connectors [Robbes 2005]. The survey of versioning systems for software evolution research [Robbes 2005] suggests that the granularity of information contained in versioning systems is not complete enough to perform higher quality evolution research. In versioning systems, source code changes are captured using the source code commits by developers that may impose the following limitations:

1. The time between two commits varies widely - often as much as several hours or days - to maintain the changes between two source files. What changes happen between two commits is most often not stored in the versioning system. This results in a coarse-grained and usually degraded information as far as capturing details of architecture evolution is concerned.

2. The commits are done usually at the developer's will, therefore several independent changes (on source files) can be introduced in one single commit, making it hard to distinguish between the individual changes and maintaining change granularity.

Since the past evolution of a software system is not a primary concern for most developers, it is not an important requirement when designing versioning systems [Zimmermann 2005, Gall 1997]. However, the details of information stored in a change log can be exploited to capture fine grained instances of change operations over individual architecture elements. In order to provide an experimental foundation for evolution analysis, architecture change log provides source of evolution knowledge that can be extracted with change mining. We discuss the role of change logs in maintaining a history of architecture evolution and a source repository for architecture change investigation in Chapter 5.

Collection II - Language as a Collection of Change Patterns

The potential beyond individual patterns is realised as a collection of change patterns that represent a generic and potentially reusable solution to a set of evolution problems. A language-based approach [Goedicke 2002] provides a vocabulary and grammar that focuses on the pattern relationships that build on each other to formalise a generic problem-solution view to enable reusable evolution. As an integrated solution, in Figure 4.2 we propose change mining to empirically derive an explicit reusable knowledge as the pattern language that represents a formalised collection of change patterns. We discuss a pattern language as a pattern collection in Chapter 8.

4.4 A Comparison Summary of Existing and Proposed Solution

Before conclusions, we now provide a comparison of the overall proposed solution to the most relevant research on architecture evolution reuse. The studies [Barnes 2013, Yskout 2012, Goedicke 2002] are considered most relevant (based on the systematic review, cf. chapter 3) as they are specifically focused on pattern-based evolution of architectures. In software architecture community, pattern oriented software architecture [Buschmann 1999] represents one of the foundational literature on patterns and pattern languages for architecture design. In contrast to patterns of architectural design in [Buschmann 1999], our solution is the first attempt towards promoting an empirically derived pattern language to enable reuse in architectural evolution.

1. Reusable Evolution Plans and Patterns - in [Barnes 2013, Yskout 2012] the research has focused on exploiting reusable plans and patterns to evolve software architecture. More specifically, [Barnes 2013] highlights a plan-based method to derive various *evolution paths* for an architecture that can be reused. Moreover, [Yskout 2012] presents patterns for *co-evolution* of the requirements and the architecture. These solution rely on plans and patterns that are derived based on individual experiences rather than an empirical and continuous discovery.

   In contrast [Yskout 2012]our solution is limited to supporting architectural evolution and do not support co-changes in requirements and their corresponding architecture model. With our solution illustrated in Figure 4.2, our solution is not limited to pattern-based change execution, it also supports change mining for pattern discovery.

2. Pattern Language for Architectural Migration - in [Goedicke 2002] 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 for migrating C language implementations to components in an object system. Our solution is not focused on migration of legacy code to components, instead it supports reuse of architecture evolution. We propose that change patterns as generic reusable abstractions must be empirically identified as recurring, specified once, and instantiated multiple times to benefit evolving architectures. With pattern-based change management, our solution promotes a semi-automated selection of appropriate patterns with necessary user intervention [Goedicke 2002].

Chapter 5

Change Logs as a Source of Architecture-centric Evolution Knowledge and Pattern Discovery

Contents

5.1 Chapter Overview

Considering the role of collections in the PatEvol framework (Chapter 4), this chapter is focused on exploiting change logs as a repository infrastructure for maintaining and analysing architectural changes. The primary intent of this chapter is to define architecture change logs as a source of evolution-centric knowledge and a repository infrastructure for architecture change mining.

A change log represents *'an explicit source of evolution-centric knowledge that maintains and provides a sequential collection of architecture change history that has been aggregating over-time'* [Yu 2009, Lassing 2003]. This chapter focuses on:

- *Capturing Architecture Change Instances in the Log* - The first step towards architecture evolution analysis includes capturing the architecture change instances in the log.
- *Classifications of Architecture Change Log Data* - Once we capture the change instances in the log, it is vital to distinguish between different types of data in the log. The data in the change log is classified as *change data* and *auxiliary data*.
- *Identification of Architecture Change Sessions* - The identification of change sessions is vital in order to create the subsets of change log data. Change sessions in a log allow us to analyse changes from a different point-of-view (e.g: time, type, and user of change, etc.).
- *Creating a Change Log Graph* - The final step includes the representation of change log data as a graph. Based on graph-theoretic details in Chapter 2, we exploit attributed typed graphs for log data representation.

The outcome of this chapter is a formalised graph-based modelling of architecture change representation that results in a change log graph. In this chapter, we aim to address **RQ 1** (cf. Chapter 1) that highlight the needs for a modelling notation that supports a formal representation and analysis of architecture evolution histories.

5.2 Change Logs as Source of Evolution Knowledge

Change logs provide an explicit source of evolution-centric knowledge as a structured collection of architecture change sequences. These change sequences represent addition, removal or modification of architecture elements that causes architecture evolution. Therefore, an individual architecture change represents the most fundamental unit of architecture evolution.

Evolution-centric knowledge in the change logs is represented as a sequential history of architecture changes as presented in Figure 5.1. For example, in Figure 5.1 Source to Target architecture evolution includes a number of intermediate architectural changes (A_{CN}) expressed as $Source \xrightarrow{Evolution} Target :< A_{C1}, A_{C2}, A_{C3}, \ldots, A_{CN} >$, where $Source \xrightarrow{Evolution} Target$ represents the path of evolution as detailed in [Garlan 2009].

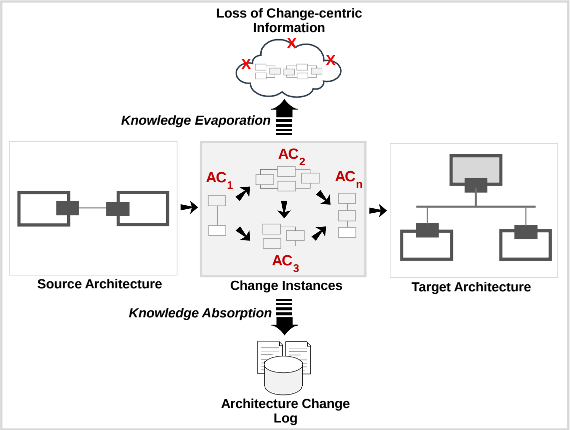


Figure 5.1: Capturing Architectural Change Instances during Evolution.

Once a sequential collection of architectural changes is maintained, we could perform the post-mortem analysis on the architectural evolution history [Gîrba 2006] to perform fine-grained history analysis [Bengtsson 1999]. This evolution-centric knowledge is represented in the form of a *taxonomical classification of change operations*, *operational dependencies*and *architecture change patterns*. In the context of Figure 5.1, if change instances are not explicitly captured this results in the loss of change-centric information that we refer to as the *evolution-centric knowledge evaporation*. In contrast, capturing each individual architectural change enables maintenance of a fine-granular representation of architecture evolution history in a repository infrastructure we refer to as *knowledge absorption*. Such an absorbed knowledge from architecture evolution process or an evolution path [Garlan 2009] provides us with an experimental foundation to maintain and analyse a historical view of architecture evolution.

5.2.1 Architecture Change Instance vs Architecture Change Operation

In literature, the terminologies a) architecture change instance(s) and b) architecture change operation(s) are often used interchangeably as both refer to architectural changes [Williams 2010, Tamzalit 2010]. However, for the sake of a technical clarification we must distinguish between the following:

- *Architecture Change Instance* - is a general reference to an individual change applied to an architecture model. For example, add an element C of type component refers to an instance of architectural change.

- *Architecture Change Operation* - provides operational details for a formal representation of a change instance. For example, an operational representation of a change instance includes the name of change operation (Add()) and its parameters representing the architecture element and its type (C hasType Component). Adding an element C of type component is operationally expressed as: Add($C \in CMP$).

5.3 Recording Architecture Changes in Logs

During the change mining process, representation of architecture changes (cf. Figure 5.1) in a log is fundamental to performing any analysis on change log data. To capture change log data, first we present a meta-model of the change log. The meta-level information is vital to identify a structural representation of change log data in Section 5.3.1. In addition, log meta-model also helps us to determine the structural representation of a change log graph (what defines a node, what are the edges, etc.). We discuss representation and classification of log data in Section 5.3.2.

5.3.1 A Meta-model for Architecture Change Logs

The meta-model for an architecture change logs is derived based on the representation of changes in a log as a constrained composition of the change operationalisation on architecture elements in Figure 5.2. The details of information stored in a change log depend on the granularity of change itself that may vary from a simple change like adding a port to a component that involves a single change operation. In contrast, a sequential combination of individual change operations may result in a more complex change like integrating a new component in existing architecture that involves multiple change operations and their cascading effect on architecture composition [Tu 2002, Bengtsson 1999]. A change log meta-model is composed of:

- **Entities** that represent a core element of log data. An example of a log entity is a Change Operator that is applied to another entity Architecture Model.

- **Entity Group** that organises a set of related entities into a logical grouping. This grouping is represented as auxiliary data and change data.
- **Entity Relations** refer to three types of relations as composition, generalisation, association among elements of log data.
  - *Composition type relation* - refers to the part-whole relation based on the atomic and composite entities. More specifically, a composite entity is composed of one or more atomic entities. For example, in the architecture model in Figure 5.2 the configuration is a composite entity that is composed of a component(s) that itself is composed of port(s). Extended details about architectural composition are provided in Chapter 2.
  - *Generalisation type relation* - refers to the generic-specialised relation among two or more entities in the log model. For example, in Figure 5.2 a *change operator* is a generalised concept of the more specific operations (e.g: Add, Remove and Modify).
  - *Association type relation* - refers to a possible association among two or more entities in the log meta-model. For example, an association relation is expressed as a change operator is *appliedTo* architecture model as presented in Figure 5.2.

It is of central importance to provide a mechanism that enables capturing fine-grained change representation along with flexible mechanism to store and retrieve the change information. In Figure 5.2, based on the internal structure of change log (entities and their grouping), log data can be classified as *change data* and *auxiliary data* in Figure 5.2. The change data in Figure 5.2 represents the core of evolution-centric information in terms of change operations on the architecture model and constraints on change operations. These constraints ensure structural integrity of the architecture model before and after architecture evolution. In addition, the auxiliary data in Figure 5.2 captures the intent, scope, time and user (person) who applied the change. The auxiliary data represent User ID, Time Stamp, Change Intent and System ID. We further explain the classification of change log data in next section after clarifying the log meta-model.

Architecture Model (ARCH)

We borrow the architectural modelling from Chapter 2 with an architecture model consisting of *Configurations* that are composed of *Components* and *Connectors* containing *Ports* and *Endpoints* respectively as presented in Figure 5.2. More specifically, we represent a component-based architecture model as topological *configurations* based on a set of architectural *components* (containing

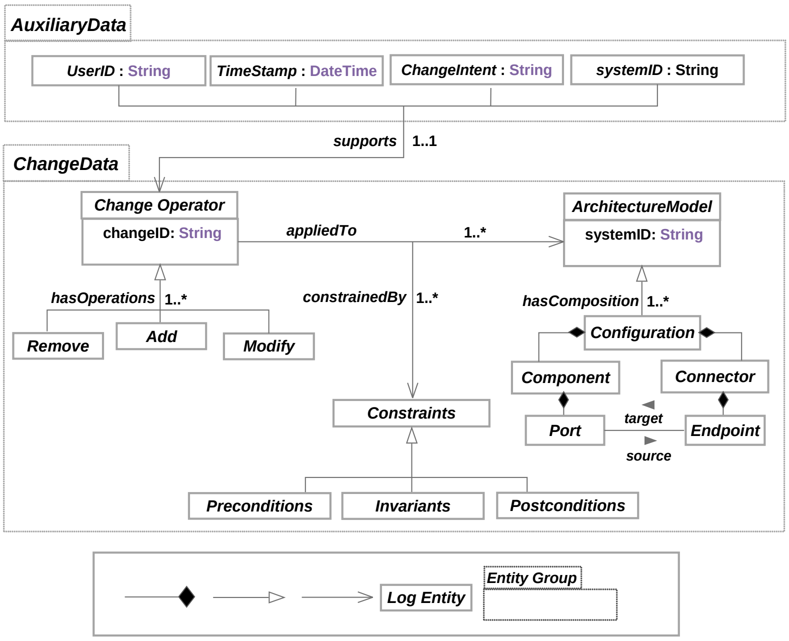


Figure 5.2: A Metamodel Representation of the Architecture Change Logs.

ports) as the computational entities that are linked through *connectors* (that connect component ports using endpoints) [Medvidovic 1999, Garlan 2009] in Figure 5.2. This description of an architecture model can be extended to add further elements. For example a possible extension could involve specifications of components operations exposed on a given port, while an endpoint can have binding among operations (if required but currently out of scope for this research). The log meta-model only captures architectural changes that conform to architecture model in Figure 5.2. This means, more traditional object-based architectures - inheritance and aggregation relations - needs some modifications in the log structures for representation addressed in [Tu 2002].

The inheritance and aggregation type relations are typical to object-oriented systems in order to promote generalisation-specialisation type classes and their objects. However, a comparison of the object vs component based systems [Erl 2009b] suggests that these relations also introduce a tight coupling between the generalised and specialised (also called *parent* and *child*) type objects. In contrast, one of the main characteristic of component-based development is to minimise (or ideally eliminate) such tighter coupling [Szyperski 2002, van der Aalst 2002] for developing reusable off-the-shelf components. This is achieved by exploiting the concept of component composition (avoiding any inherited properties of a component). Therefore, if we consider capturing changes for object-oriented systems; the concerns for such (inheritance and aggregation) relations must be explicitly addressed for change implementation and change analysis [Tu 2002].

Change Operator (OPR)

Architecture change operators represent *addition*, *removal*, and *modification* type changes on architecture models such that Change Operator is *AppliedTo* Architecture Model in Figure 5.2. A change operator provides an operational - operator name and its parameters - representation and abstraction for architectural changes. We provide details about the syntax and composition change operations later in the thesis. However, for the sake of clarification an operation $Add(P_m \in PORT : C_n \in CMP)$ represents addition of a port (P_m) to an existing component (C_n), $\in$ represents element type relation (C_n is of type CMP).

Constraints (CNS)

The constraints represent a set of conditions on architecture change operations expressed as *Preconditions*, *Invariants* and *Post-conditions*. During change operationalisation pre-conditions represent the structural composition of architectural model as well as individual elements before change execution. It represents a complete or partial source architecture model that is evolved towards a target model. Continuing with our previous example of adding a port P_m in component C_n, preconditions ensures a) a component C_n already exists in the architecture model and b) a port P_m does not currently exist in the component. After change operationalisation the post-conditions represent the evolved architecture model or an individual element as a consequence of applying change operationalisation. For example, postconditions ensure that a port has been successfully added into the component such that C_n contains a new port P_m.

5.3.2 Log-based Representation of Architecture Change Instances

We explain recording of the individual architectural changes in the log with the help of the Electronic Bill Presentment and Payment architecture evolution case study [EBPPCaseStudy]. Architectural representation for EBPP and details about the selection of its evolution scenarios are presented in **Appendix B**. We adopt the Architecture Level Modifiability Analysis (ALMA) [Bengtsson 1999] method for evolution scenario elicitation and analysis of EBPP architecture evolution. We follow the ALMA method for selection, evaluation and interpretation of the evolution scenario in Figure 5.3.

Evolution Scenario Selection - Component Integration

We present the evolution scenario of component integration in the EBPP case study. More specifically, in the existing functional scope of the case study the company charges its customer with full payment of customer bills in advance to deliver the requested services. Now, the company plans to facilitate existing customers with either direct debit or credit-based payments of their bills represented in Figure 5.3. In Figure 5.3, this evolution scenario is represented as: *integration of a mediator component PaymentType that facilitates the selection of a payment type (direct debit, credit payment) mechanism among the directly connected components BillerCRM and CustPayment.*

Evolution Scenario Evaluation - Analysing Architectural Changes

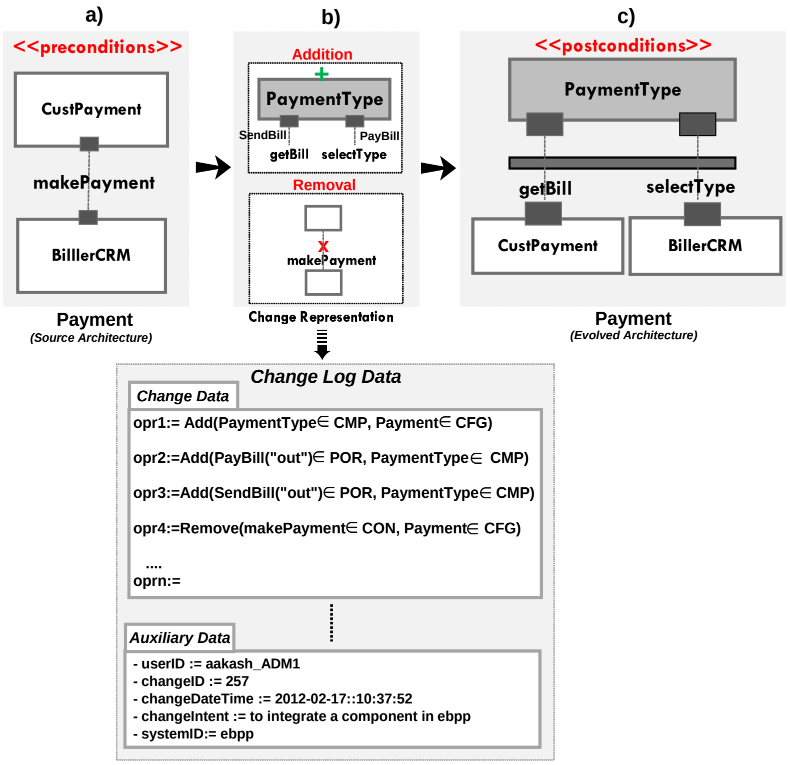
Once the evolution scenario is selected, we are interested in analysing the architectural change operations that are applied to architecture elements to execute this scenario. Furthermore, the change operations are captured in the change log for post-mortem analysis of architecture evolution scenarios. In the case of component integration, the EBPP architecture is modified with an addition of new components PaymentType and two connectors getBill and selectType to mediate customer billing and payments, represented in Figure 5.3.

Results Interpretation - Impacts of Change of Architecture

We interpret the results of a given evolution scenario based on the impact of architecture changes on an existing architecture as illustrated in Figure 5.3. This is represented as the source architecture (as preconditions of evolution), the architectural changes (as change operations) applied to a source architecture to obtain the evolved architecture (as post-conditions of evolution).

1. *Change preconditions*: The existing configuration consists of a direct interconnection makePayment between the components CustPayment and BillerCRM that are represented as a change preconditions in Figure 5.3 a).

2. *Change Operations*: In order to integrate the new functionality that enables the selection of a payment type option for customer payments, this change is represented in Figure 5.3 b) as the addition of a PaymentType component in the Payment configuration. This results in five changes as recorded in the change log and illustrated in Figure 5.3. These changes include the addition of the component ($opr1$), its ports ($opr2$, $opr3$) and removal of the old connector makePayment ($opr4$).



```
opr1:= Add(PaymentType∈ CMP, Payment∈ CFG)
opr2:=Add(PayBill("out")∈ POR, PaymentType∈ CMP)
opr3:=Add(SendBill("out")∈ POR, PaymentType∈ CMP)
opr4:=Remove(makePayment∈ CON, Payment∈ CFG)
 ....
oprn:=
```

```
- userID := aakash_ADM1
- changeID := 257
- changeDateTime := 2012-02-17::10:37:52
- changeIntent := to integrate a component in ebpp
- systemID:= ebpp
```

Figure 5.3: Representation of Auxiliary Data and Change Data in Logs.

3. *Change post-conditions*: The application of the changes results in an integration of the PaymentType component that mediates the selection of payment type among CustPayment and BillerCRM components, presented as the change post-conditions in Figure 5.3 c).

The role of preconditions is to ensure that elements to be added do not already exist in the log. Post-conditions ensure elements have been successfully added to the change log.

Definition 5.1. Architecture Change Log - Let OPR_i represent an individual change operation. An architecture change log (ACL) is a sequential collection of change operations expressed as a tuple $ACL =< OPR_1 \prec OPR_2 \prec \ldots \prec OPR_N >$. $\prec$ is a sequencing operation between change operations (OPR_1 to OPR_N).

A change log represents a sequential collection of individual change operations on architecture elements. For example, in Figure 5.3, change operations $(opr_1, opr_2, \ldots, opr_n)$ are represented as a sequential collection of architectural changes (Add, Remove, Modify) on architecture elements (components, connectors, configurations).

Once sequential architectural changes are represented and recorded in change log (Definition 5.1), change log data is classified as *Change Data* (CD) and *Auxiliary Data* (AD) in Figure 5.4.

1. *Auxiliary Data (AD)*: provides the additional details about individual change instances in the log. This is expressed as $AD :=< userID, changeID, DateTime, changeIntent, sytemID >$ and is captured automatically along with some user input in Figure 5.4 as detailed below.

2. *Change Data (CD)*: contains the core information about individual change instances in the log. This is expressed as $CD =:< cID, Opr, ArchElem, ElemType >$ representing change id $(opr_1, opr_2, \ldots, opr_n)$, along with change operations (Opr) on architecture elements ($ArchElem$) that has a type ($ElemType$) from Definition 5.1.

- *Capturing the Auxiliary and Change Data in the Log*: in Figure 5.3, we illustrated a scenario-driven approach (guided by ALMA [Bengtsson 1999]) to represent the architectural changes in the log. After presenting the types of data in the log, we briefly discuss the process for capturing data in the log.

  - *Capturing Auxiliary Data* - the elements of the auxiliary data that include user id (Aakash-ADM1), change id (257), date-time (10:37:52/17/02/2012) and the system identifier (EBPP) are captured automatically as soon as a change is applied (cf. Figure 5.3). However, the intent of the change (e.g: to integrate a component in EBPP) must be specified by the user to explicitly represent *what was the need for this change*? Different users may have different intents of a change, for example the removal of a component from the architecture is permanent or it has been removed due to a replacement. The auxiliary data is particularly useful for architectural change analysis based on the source, intent, time of change and facilitates in extracting specific (time/user-based etc.) architecture change sessions from log - detailed in Section 5.4.

  - *Capturing Change Data* - in contrast to the auxiliary data, capturing change data is automatic and no user intervention is required. For example, in Figure 5.3, $op1$ represents addition of a new component PaymentType inside the Payment configuration that is recorded as an individual change in the log. Change Data helps us to create the change log graph that is detailed in Section 5.5.

To maintain a fine granular representation of architectural changes, we investigate six aspects in change log data as presented in Figure 5.4:

- **Who** performed a specific change in existing architecture model (Person/Architect responsible for changes)

- **When** a specific change is performed (Time-Date of a specific change),
- **Why** the change was performed (Intent and Rationale for change provided by the architect)
- **What** is effect of change on architecture elements (Change Operations on Architecture Elements),
- **Where** a particular change is applied in an existing architecture model (Parameters of Change Operations)
- **How** to locate a particular change in a collection of changes (log) (Maintaining Change Session and Change Id)

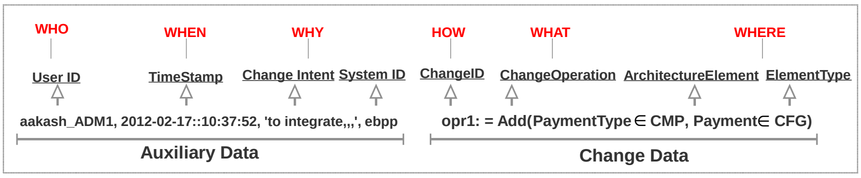


Figure 5.4: Representation and Classification of the Change Log Data.

5.4 Preserving Evolution History in Change Logs

Architecture change logs are characterised as a sequential collection of changes that represent a history of architectural changes [Yu 2009]. In the history-centred approaches for change mining, change history represents an ordered set of change versions with added information about the time of change [Zimmermann 2003, Gîrba 2006]. In preserving evolution history, the primary intent is to enable future analysis focused on *when some change happened* and also to analyse *what was the change impact* along with its scope and intent [Gîrba 2006]. The main idea behind the maintenance of history is to analyse architecture evolution according to a particular point of analysis that is also referred to as an architecture change session.

The intent and example for each of the session is detailed as below. Specifically, the Change-based session is used by us to analyse different types of change operations (detailed later in the thesis). The other two types of session are not used in this thesis, however; they represent a customisation for future needs, if required to investigate changes in terms of *specific time interval* or *specific person* responsible for the change. Therefore, a brief discussion of these change sessions exemplify as well as highlight possible variations, customisation or future extensions based on the type of required change analysis.

5.4.1 Maintaining Architecture Change Sessions

A change session represents a predefined (time, user, change based) subset of all the changes that are recorded in the change log. Session-based analysis of change representation is particularly beneficial to analyse the time, intent, scope and operationalisation of changes.

Definition 5.2. Architecture Change Sessions - An architecture change session CS in the log (Definition 5.1), is represented as tuple: $CS = < User, Time, OPR >$:

- **User** represents a change session based on all the changes performed by a specific $userID$.
- **Time** represents a change session based on all the changes within a specific time interval (T_N - T_0).
- **OPR** represents a change session based on the type of a specific change operation (Add or Remove or Modify).

A summary of the different types of change session functions is provided in Table 5.1 with explanation as follows. The three types of sessions are provided to support the customisation for the analysis of architectural changes (if and whenever required) by means of parameters for each of the session function in Table 5.1.

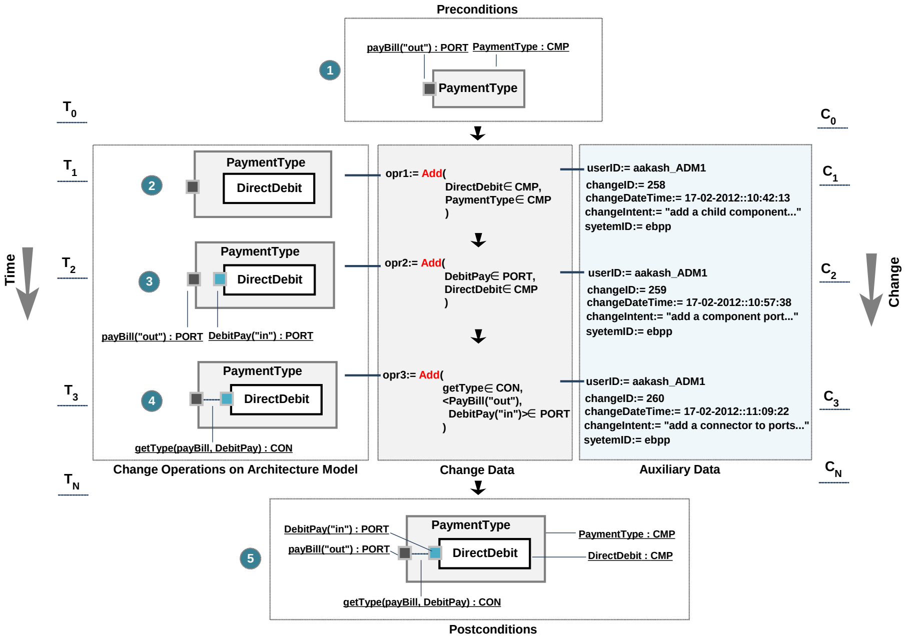
Change Session Function	**Parameter(s)**	**Return Values**
userSession(userID)	A unique user identification	All change operations performed by specific userID
timeSession(strTime, endTime)	Time interval as start and end time.	All change sessions between interval (endTime - strTime)
changeTypeSession(Opr)	Predefined change operators	All change with a specific change operation (Opr).

Table 5.1: Summary of Different types of Architecture Change Sessions.

1. **User-based Session** All the changes in the log that are performed by a specific user that is identified by unique user identification. It facilitates with analysing the changes based on an individual's intent of architecture change. For example, in Figure 5.5 the usage-based session reflects only those changes that have been performed by userID: aakash-ADM1. In other examples, all the changes that have been scattered across the change log (performed by aakash-ADM1) can be collected and analysed as a change session that is dynamically created.

2. **Time-based Session** All the changes in the log that are performed between a given time interval (*endTime* − *startTime*). It facilitates analysing the changes by adding the temporal context. For example, in Figure 5.5 the time-based session reflects only those changes that have been performed in a given interval of time (approx. 27 minutes from 17-02-2012::11:09:22 to 17-02-2012::10:42:13).

3. **Change-based Session** All the changes in the log that are performed by a given change operation (Add or Remove or Modify). It facilitates with analysing the intent, impact and scope of change with a similar kind of operationalisation (inclusion or exclusion of architecture elements). For example, in Figure 5.5 the change-based session reflects only those changes that result in an addition of architecture elements (Add()). This projects only those changes which resulted in an inclusion of architecture elements in the existing architecture model.



```
opr1:= Add(
        DirectDebit∈ CMP,
        PaymentType∈ CMP
        )

opr2:= Add(
        DebitPay∈ PORT,
        DirectDebit∈ CMP
        )

opr3:= Add(
        getType∈ CON,
        <PayBill("out"),
          DebitPay("in")>∈ PORT
        )
```

```
userID:= aakash_ADM1
changeID:= 258
changeDateTime:= 17-02-2012::10:42:13
changeIntent:= "add a child component..."
syetemID:= ebpp

userID:= aakash_ADM1
changeID:= 259
changeDateTime:= 17-02-2012::10:57:38
changeIntent:= "add a component port..."
syetemID:= ebpp

userID:= aakash_ADM1
changeID:= 260
changeDateTime:= 17-02-2012::11:09:22
changeIntent:= "add a connector to ports..."
syetemID:= ebpp
```

Figure 5.5: Sequential Representation of Architecture Change Instances in Logs.

In Figure 5.5 we provide a sample change session from an architecture change log. The session is extracted randomly based on total change instances that have been recorded over a period of 1 day (07-08-2011). A simplified example illustrates the effect of changes on architecture model over time. The example highlights the life-cycle of a component PaymentType that is added to an architecture model at time interval T0 and modified by adding its child component DirectDebit at interval T1. Furthermore, at interval T2 a port is added to the child component that follows an addition of a connector to map ports of a child component and its parent at interval T3.

5.5 Graph-based Modelling of Architecture Change Log Data

In previous sections, we focused on the anatomy of change log data and its classification. However, in order to systematically investigate architecture change representation; we need a formalised representation for an experimental analysis of change log data. We utilise a graph-based formalism [Jiang 2012, Ehrig 2004] in order to exploit graph-theoretical foundation for representation of change log data (cf. Chapter 2 - graph-based modelling for architecture change mining). Our preference for graph-based modelling of log data over UML 2.0 based notations is already explained in Chapter 2.

5.5.1 Creating Change Log Graph

In this section, we focus on formalising change instances in the log as an *attributed graph (AG)* with nodes and edges typed over an *attributed typed graph (ATG)* [Ehrig 2004]. Please note, an ATG in Figure 5.6 represents a meta-graph to model change log data as an AG that represents an instance-graph in Figure 5.7.

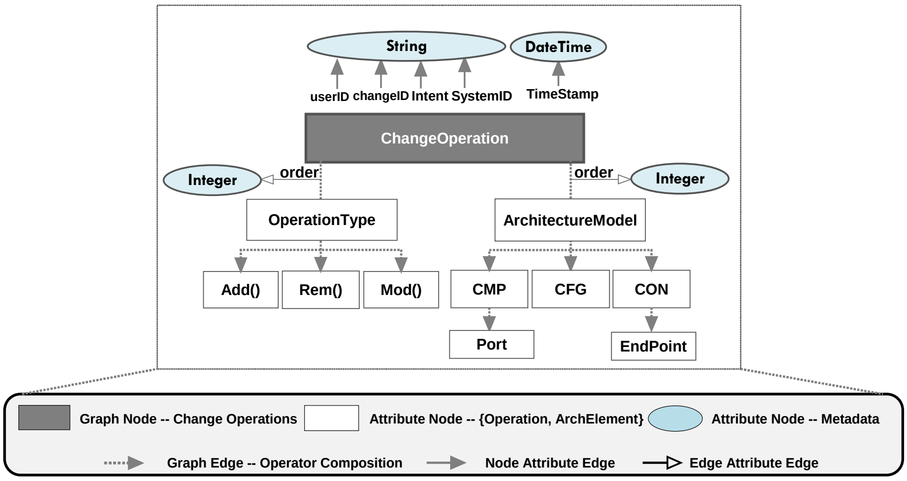


Figure 5.6: Attributed Typed Graph Model to Formalise Architecture Change Log Data.

Definition 5.3. Architecture Change Log Graph - A collection of change operations from log ACL (Definition 5.1) are expressed as an attributed change log graph G_{ACL}:

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

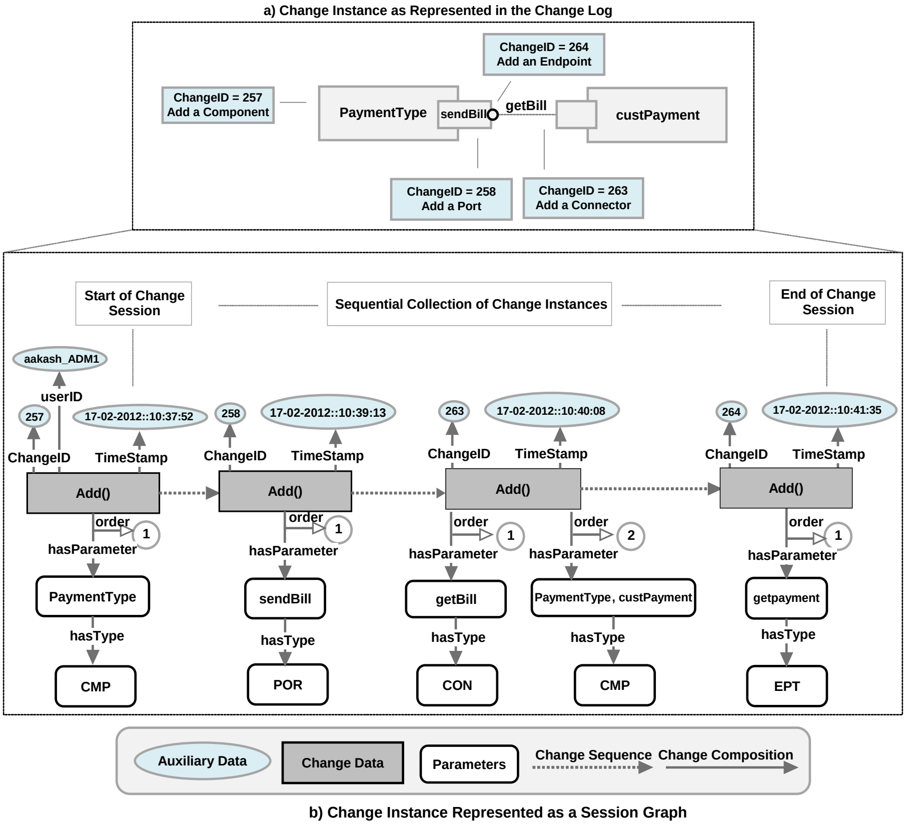
- **Graph Nodes** represent change operations on architecture model: $N_G, N_A \in Nodes$,
- **Graph Edges** represents a sequencing of nodes (operations): $E_G, E_{N_A}, E_{E_A} \in Edges$.

The attributed graph morphism M from an instance graph AG (Figure 5.7) to its meta-graph ATG (Figure 5.6) is expressed as $M : AG \rightarrow ATG$. We formalise log data as attributed nodes and edges below and exemplify a possible instance of change log graph in Section 4.2 in the context of Figure 5.7

1. $N_G = \left\langle n_g^i | i = 1, \ldots, m \right\rangle$ represents a set of graph nodes. Each graph node ($n_g \in N_G$) represents a single change log entry (i.e., a single change operation) - introduced in Figure 5.3. The sequence $i = 1, \ldots, m$ refers to the total number of change operations that exist in the log. The notation m is an upper bound (starting from 1 (first node) and leading to m (final node)) in the sequence.
2. $N_A = \langle n_a^i | i = 1, \ldots, m \rangle$ represents a set of attribute nodes for graph nodes (N_G). Attribute nodes are of two types, a) attribute nodes that represent auxiliary data (e.g. userID, changeID, TimeStamp etc.) and b) attribute nodes that represent change data and its subtypes (e.g. operation type, architecture model). The sequence $i = 1, \ldots, m$ refers to the total number of attribute nodes in change log graph.
3. $E_G = \left\langle n_g^i | i = 1, \ldots, m - 1 \right\rangle$ represents a set of graph edges that connect two graph nodes N_G. The graph edges ($e_g \in E_G$) represent the applied sequence of change operations (OPR) on the architecture model (ARCH). The term $i = 1, \ldots, m - 1$ represents the total graph edges in the log graph.
4. $E_{N_A} = \langle e_{na}^i | i = 1, \ldots, p \rangle$ represents the set of node attribute edges that join an attribute node ($n_a \in N_A$) to a graph node ($n_g \in N_G$). The sequence $i = 1, \ldots, p$ refers to the total number of node attribute edges in an architecture change log graph.
5. $E_{E_A} = \langle e_{ea}^i | i = 1, \ldots, q \rangle$ is the set of edge attribute edges that join an attribute node ($n_a \in N_A$) to an attributed edge (e_{na}). The sequence $i = 1, \ldots, q$ refers to the total number of edge attribute edges in a log graph.

5.5.2 Creating Architecture Change Session Graph

The change session graph enables the extraction of a subset of all the change instances in the log based on intent, scope or time of architectural changes. Continuing with the earlier example (addition of a PaymentType component, cf. Figure 5.4), in Figure 5.7 we present a partial view of the change session graph that is an instance of the change log graph (a.k.a. AG) in Figure 5.6. The architecture change session is calculated based on an interval (endTime - start Time) as all the changes that occurred between time stamp endTime(17-02-2012::10:41:35) and startTime(17-02-2012::10:37:52). Session-based change mining is helpful in analysing a subset of all the changes from logs (time-interval of architectural change defines change subset in this example).


a) Change Instance as Represented in the Change Log

b) Change Instance Represented as a Session Graph

Figure 5.7: Change Instances as an Attributed Graph (typed over ATG in Figure 5.6).

In Figure 5.7, an attributed graph morphism $t : AG \to ATG$ is defined over graph nodes with $t(ATG) = AG$ that results in t(ChangeOperation) = Add(), t(ArchitectureElement) = PaymentType, custPayment sendBill, getBill, getPayment and t(hasType) = CMP, CON, POR, EPT where (PaymentType, custPayment) hasType CMP, (sendBill) hasType POR, (getBill) hasType CON, (getPayment) hasType EPT. The graph nodes are linked to each other using graph edges for source and target nodes (257, 258, 263, 264) representing the applied sequence of change operations.

The partial view of time-based change session from a log is represented as: ChangeID = 257 representing the addition of a Component expressed as: $Add(PaymentType \in CMP)$, ChangeID =

258 representing the addition of a Port expressed as: $Add(sendBill \in POR)$, ChangeID = 263 representing the addition of a Connector expressed as: $Rem(getBill \in CON, (PaymentType, custPayment) \in CMP)$ and ChangeID = 264 representing the addition of an Endpoint $Rem(getPayment \in EPT)$.

5.5.3 Sequential vs Hierarchical Representation of Log Data

We have represented the log data as a sequential graph. When compared to a hierarchical or parallel representation, our preference for a sequential graph (Figure 5.8) is determined by: i) the type of architectural changes in the log (i.e., *data representation*), ii) the complexity of graph matching (i.e., *data processing*) - both discussed below.

Representation: Sequential vs Parallel Architecture Changes

In the taxonomy of software evolution [Buckley 2005] and also in a characterisation of architectural changes [Williams 2010] two major types of change are classified as *sequential* or *parallel* (a.k.a. hierarchical) changes. The sequential changes are common in an environment where a single team is responsible for change execution and management. Sequential change restricts more than one person to apply changes to the same architecture at the same time. In contrast, parallel changes are applied in a collaborative development environment where multiple teams are working on the same architecture. Therefore, multiple persons can apply changes to the same architecture at the same time. Another approach is to enforce the concurrency control (e.g, priority based change implementation), such that all parallel changes are converted into a sequence based on the priority or criticality of the change.

Assumptions and change type transformation - the process of log-based change mining for pattern discovery in this thesis is focused on analysing the sequence of architectural changes [Lehnert 2012, Sun 2010] illustrated with a (sequential) change log graph in Figure 5.7. We have assumed that parallel change operations (if any in the log) are represented as a sequence (parallel to sequential conversion [Buckley 2005]), where each of the change operations is executed one after the other (i.e., sequenced change log). Such a restriction is also inherent in our change capturing process, where only one person can apply the change on the same architecture element at a given time. In situations where parallel changes are present, the meta-model for a change log graph (Figure 5.6) must be extended to support parallelism. The assumption for sequential change analysis is also based on the work on mining sequential patterns [Agrawal 1995].

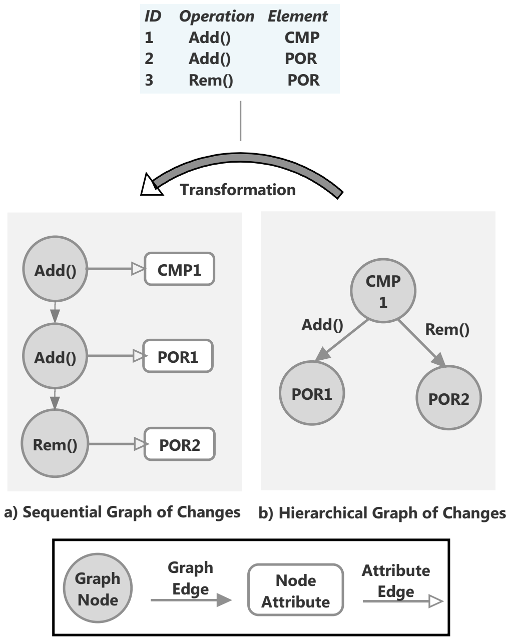


Figure 5.8: Overview of Sequential vs Hierarchical Representation of Log Data.

Processing: Sequential vs Hierarchical Graph Matching

In the context of change analysis (with graph matching), Figure 5.8 provides a high-level view of the alternative representations of log data as a) a sequential graph and b) a hierarchical graph. In the sequential representation (Figure 5.8 a)), a change log graph is constructed such as graph nodes are change operations, node attributes are architecture elements encapsulated as a parameter to the operation and graph edges represent the sequence among change operations. Alternatively, with a hierarchical representation (Figure 5.8 b)), graph nodes represent the architecture element, graph edges represent a change operations, while the hierarchy of nodes represent the composite element. The conversion of a hierarchical structure to a sequential one [Leung 2005] enables a sequential processing of data to minimise the complexity of graph matching [Agrawal 1995, Geng 2008]. Hierarchical to sequential conversion is also referred to as transformation - (also parallel to sequential conversion [Buckley 2005]) in Figure 5.8.

Minimising Complexity of Graph Matching - graph-based modelling of change log data allows us to utilise the frequent sub-graph mining approach to discover recurring sequences (sub-graphs) as change patterns. However, discovering patterns by matching and mining sub-graph is a complex problem that is known to be NP-complete [Conte 2004] and it is not known whether pattern discovery using graph mining is possible in a polynomial time. Therefore, a simplified representation of the change log graph (without compromising the log data representation) helps in minimising this complexity. Specifically, considering the context of Figure 5.8 b), if an architecture element is represented as a node and a change operation as an edge, we need to match both the nodes and edges to analyse i) what architecture element is affected (e.g., component and connector, etc.), and ii) what was the change (e.g., add or remove, etc.). This results in a significant increase of the complexity of even simpler cases, since both node and edge matching is required to interpret an individual change. In contrast, Figure 5.8 illustrates a sequential graph in which an edge is simply a sequence between two change operations (nodes) for graph traversal and the matching of edges is not required.

In a sequential graph only nodes (operations) needs to be matched. In case of hierarchical graph, matching both nodes and edges of a graph (graph isomorphism) increases the complexity of even simpler cases, where both nodes and edges need to be matched. In comparison to the hierarchical graph matching, the algorithmic solutions to discover sequential patterns offer better performance (with reduced complexity and faster matching) [Huang 2003, Conte 2004]. In our case, in addition to the design simplicity of pattern discovery algorithms, sequential graphs helps with the reduction of graph matching complexity.

5.6 Mapping Log Data to GraphML-based Representation

A change log graph in Figure 5.7 only represents a conceptual model that needs a concrete description to automate graph-based pattern discovery. Listing 5.1 shows a GML-based representation [Brandes 2002a] of an architecture change log graph describing the addition of a new component (PaymentType : node id 257) that is connected to an existing component with addition of a new connector (getBill : node id 258) from Figure 5.7. In order to represent an explicit intent of change, we need to define the meta-level attributes (as ATG represented in Lines 7 and 12 in Listing 5.1, for space reasons extra attributes omitted). Auxiliary data attributes capture the intent, time, user of change along with id of the system (EBPP [EBPPCaseStudy] in this case) to which the change is applied (Line 14 - 18). Change data attributes capture change operationalisation on architecture elements that is needed to define the meta-level attributes (as ATG represented in Lines 19 - 23).

Listing 5.1: GraphML-based Representation of Change Log Data

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
```

```
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
              "xsi:schemaLocation="http://graphml.graphdrawing.org/
               xmlnshttp://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

    <!-- Graph generated by sones GraphAPI GraphMLWriter -->
       <key id = "Intent" for = "node" attr.name = "Desc" attr.type = "string"> </key>
       <key id = "TimeStamp" for = "node" attr.name = " TimeStamp" attr.type = "string"> </key>
       <key id = "userIDr" for = "node" attr.name = "Committer" attr.type = "string"> </key>
       <key id = "systemID" for = "node" attr.name = "sysID" attr.type = "string"> </key>
       <key id = "Opr" for = "node" attr.name = "opr" attr.type = "string"> </key>
       <key id = "hasParam1" for = "node" attr.name = "hasParam1" attr.type = "string"> </key>
       <key id = "Param1Type" for = "node" attr.name = "Param1Type" attr.type = "string"> </key>
       <key id = "hasParam2" for = "node" attr.name = "hasParam2" attr.type = "string"> </key>
       <key id = "Param2Type" for = "node" attr.name = "Param2Type" attr.type = "string"> </key>
       <key id = "Seq" for = "edge" attr.name = "type" attr.type = "composition"> </key>

          <graph id="ChangeGraph" edgedefault="directed">    <!-- Log Graph Definition -->

              <node id = "257">    <!-- Change Operationalisation - Attributed Nodes -->
                  <data key="Intent"> Add a Component </data>
                  <data key="TimeStamp"> 17-02-2012::10:37:52 </data>
                  <data key="userID"> aakash_ADM1 </data>
                  <data key="systemID"> EBPP </data>
                  <data key="Opr"> ADD </data>
                  <data key="hasParam1"> PaymentType </data>
                  <data key="Param1Type"> CMP </data>
                  <data key="hasParam2"> </data>
                  <data key="Param2Type"> </data>
              </node>

               <node id = "2">    <!-- Change Operationalisation - Attributed Nodes -->
                  <data key="Desc"> Add a Port</data>
                  <data key="TimeStamp"> 17-02-2012::10:39:13 </data>
                  <data key="UserID"> aakash_ADM1 </data>
                  <data key="systemID"> EBPP </data>
                  <data key="opr"> ADD </data>
                  <data key="hasParam1"> sendBill </data>
                  <data key="Param1Type"> POR </data>
                  <data key="hasParam2"> </data>
                  <data key="Param2Type"> </data>
               </node>

             <edge id ="e256" source = "257" target= "258"> !-- Change Sequencing - Attributed Edge -->
             </edge>

     </graph >

  </graphml>
```

5.7 Chapter Summary

The primary contribution of this chapter is to introduce the architecture change logs [Yu 2009, ROS-Distributions 2010] as a sequential collection (history) of architectural changes that can be analysed and represented in a formal way using graph models [Brandes 2002a, Ehrig 2004]. This chapter addresses **RQ 1** (cf. chapter 1) that highlights the needs for a modelling notation that supports representation and analysis of architecture evolution histories.

This chapter provides a foundation to exploit repository mining techniques on change logs to discover explicit evolution-centric knowledge as part of the architecture change mining process. We claim that if change logs represent a source of evolution-centric knowledge, we can systematically investigate logs to discover such evolution-centric knowledge that can be reused to guide future architecture evolution. We conclude this chapter by modelling change log instances as an architecture change log graph that is represented using the graph modelling language (.GML notation). Graph-based modelling [Bhattacharya 2012] is particularly beneficial to model and analyse significantly large data-sets in an efficient manner.

Chapter 6

A Taxonomical Classification and Definition of Architecture Change Operations

Contents

6.1 Chapter Overview

Change operations are fundamental to the evolution of a software system and its underlying architecture models [Williams 2010]. An individual change operation (cf. Chapter 5) represents the most fundamental unit of evolution in terms of addition, removal or modification of functionality in various software artefacts including source code and architectural components [Buckley 2005, Lehnert 2012, Van der Westhuizen 2002, Ducasse 2009]. The change characterisation scheme [Williams 2010] systematically classifies different types of architectural changes. The proposed characterisation works as a decision tree to provide support for system developers to assess the feasibility of desired changes. To analyse architectural evolution we need a fine-granular representation of architecture change operationalisation that is currently missing. In this chapter, a systematic analysis of architecture change representation (with change log graph) and its operationalisation is presented that provides a foundation to discover recurring sequences as architecture change patterns. This chapter along with Chapter 7 aims to answer **RQ 2** (cf. Chapter 1) that highlights the needs for analysing evolution histories to discover reuse knowledge.

6.2 A Taxonomy of Architecture Change Operationalisation

In order to study change representation from logs and to ultimately discover change patterns, we taxonomically classify architecture change operations as illustrated in Figure 6.1. Before a discussion of the operational taxonomy, it is vital to discuss a) *what are the needs for a taxonomy* and b) *how the taxonomy is developed and utilised to support architecture evolution knowledge*.

6.2.1 The Needs for Operational Taxonomy of Architectural Evolution

In recent years, the studies and reviews of architectural evolution research focused on evolvability analysis [Breivold 2012] and change characterisation [Williams 2010]. These and other studies like [Buckley 2005] identified that, in order to support a systematic identification and investigation of evolution, different types of changes and their impact on architecture should be classified. Classification of the individual changes provides a systematic analysis of evolution. More specifically, in comparison to analysing a classified representation of source code changes [Lehnert 2012], there does not exist any work that distinguishes different types of architecture change operations and their role in architecture evolution. The existing research has supported a *taxonomy of architectural maintenance* [Ducasse 2009], and *characterisation of architecture changes* [Williams 2010]. However, based on the guidelines to classify software changes [Buckley 2005] in general and architectural maintenance and evolution [Ducasse 2009] in particular, we derive a taxonomy of change operations.

The taxonomy provides a systematic representation of the different types of architectural changes to investigate architecture evolution. Our systematic review of the research on architecture evolution (cf. Chapter 3) highlights that: the elements of a fine-granular representation of architecture-centric changes include but are not limited to: a) *types of architectural changes*, b) *syn-*

tactical descriptions to represent these changes, and c) *various types of dependencies among these changes*. We have defined the change operation taxonomy as: *a systematic identification and organisation of different change operations into groups which share, overlap or are distinguished by various attributes such as operational composition, their representation and dependencies.*

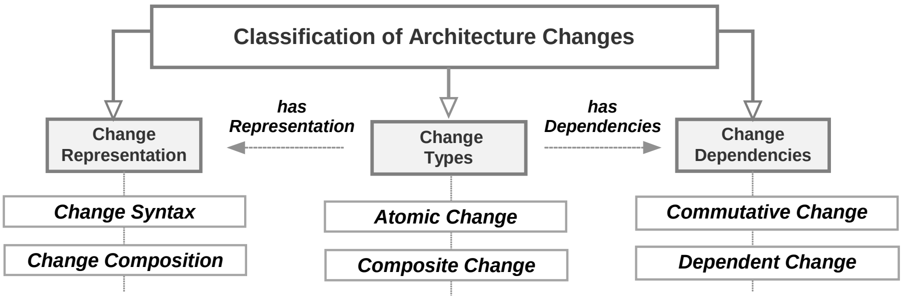


Figure 6.1: Overview of the Taxonomical Classification of Architecture Change Operationalisation.

6.2.2 Types, Representation and Dependencies of Change Operations

Based on the analysis of change log graph data (detailed in subsequent sections), we present different types of architectural changes as in Figure 6.1. The change taxonomy in Figure 6.1 also guides the discussion of the composition, representation and dependencies of change operations that is the focus of this chapter.

- *What types of change operations exist in a change log during architecture evolution?*

  We analyse the different types of operations based on the composition of the architectural evolution in terms of the **atomic** and **composite** types of changes.

- *What is the necessary syntax and composition to represent different types of change operations?*

  After distinguishing the types of changes, we need to represent these changes in a consistent manner. Therefore, a systematic representation of architectural changes requires the analysis of **syntax** and the **composition** of the change operations.

- *What are the dependencies that exist between different types of change operations?*

  Once the changes are classified and represented, we also need to analyse the types of dependencies that exist among different operations. These operational dependencies are classified as **commutative** and **dependent** type change operations.

6.3 Types of Architectural Changes

In the change logs, architecture changes are classified into three distinct types. This classification includes 1) *Atomic Changes*, 2) *Composite Changes* and 3) *Sequential Changes* as presented in Figure 6.2. In Figure 6.2, we present a bottom-up (layered overview) of architectural change types expressed as OprName(ArchElement) also presented in [Javed 2009]. It is vital to mention that the evolution styles proposed by [Garlan 2009] classify addition, removal and modification of component and connectors as primitive changes, while higher order changes like component composition are represented as style/pattern-based changes - patterns as recurring sequential change [Agrawal 1995]. In Figure 6.2, atomic and composite operations (primitive changes) can be abstracted into pattern-based architecture change execution.

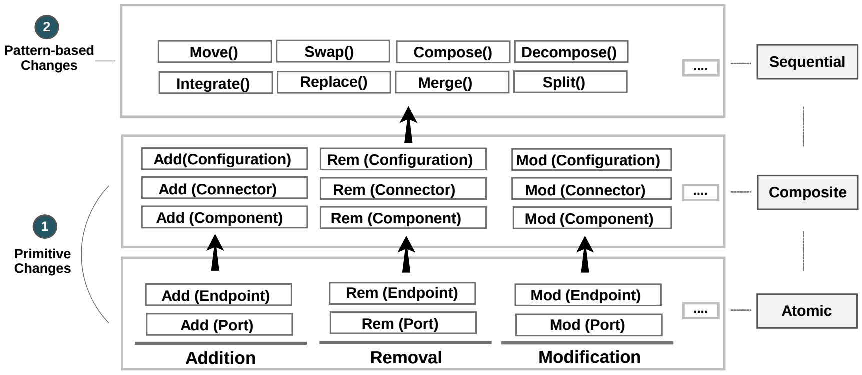


Figure 6.2: Classification of Architecture Change Types (Layered Overview).

The distinction between atomic and composite change types is determined by the composition hierarchy of the CBSA model [van der Aalst 2002, Medvidovic 2000]. Atomic change refers to changes on atomic architecture elements such as ports and endpoints. Composite change refers to changes on composite architecture elements including components (composed of ports) and connectors (composed of endpoints). Composite changes differ from the atomic ones based on explicit constraints that preserve architectural composition during architecture evolution.

6.3.1 Running Example of Architectural Changes

The example is continued from Chapter 5. We use the extracted change sequence from the change log represented as a change log graph in Figure 6.3. In Figure 6.3 a) we illustrate individual changes on the architecture model as recorded in the change log, while its corresponding log graph representation is provided in Figure 6.3 b). In Figure 6.3, we create the change session graph based on architectural changes with a time interval (*startTime* [17-02-2012::10:37:52] to *endTime* [17-02-2012::11:34:56]) as timeSession(startTime, endTime). Please note that for illustrative reasons, the change log graph in Figure 6.3 represents only necessary graph elements instead of a full representation. We only present the change id, change operators and their parameters (architecture model being changed) along with the cascading impact of change operations on the architecture model.

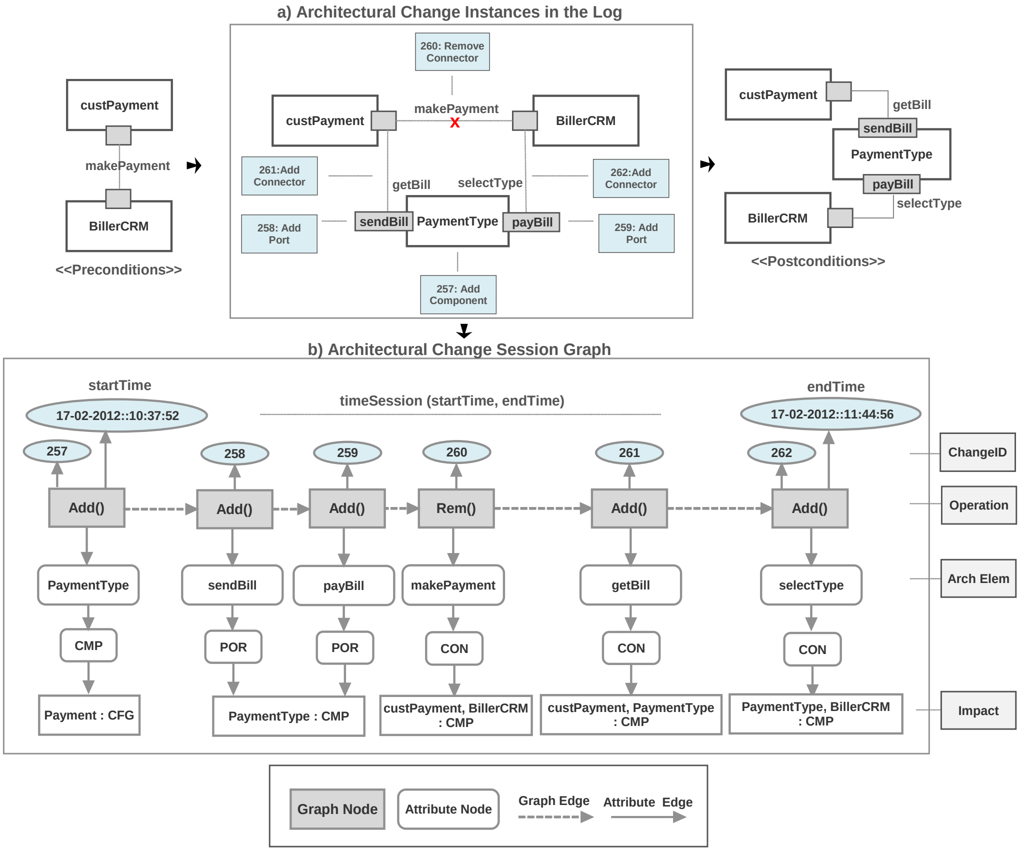


Figure 6.3: Architecture Change Session Graph (*endTime - startTime).*

In the change session graph, architectural changes denote the integration of a mediator component PaymentType with directly connected components CustPayment and BillerCRM. The preconditions of change represent components CustPayment and BillerCRM that are inter-connected using makePayment connector. The change operationalisation on architecture elements is represented as an application of the following change operations that include:

- *ChangeID = 257*: Add a new component PaymentType inside a configuration Payment.
- *ChangeID = 258, 259*: Add the corresponding ports sendBill and payBill to the newly added

component PaymentType. In this case, *ChangeID (258, 259)* can be applied in parallel referring to a commutative change, while *ChangeID (257)* precedes port addition that refers to a dependent change and specified as $ChangeID\{257 \prec (258 \parallel 259)\}$. This can be interpreted such that ChangeID 257 must be executed first, while the 'sequence of application' with ChangeID 258 and ChangeID 259 does not matter (further details in Section 6.5).

- *ChangeID = 260*: Remove the connector makePayment that connects CustPayment and BillerCRM components.

- *ChangeID = 261, 262*: Addition of two new connectors i) getBill to interconnect CustPayment and PaymentType and ii) selectType to connect PaymentType and BillerCRM.

6.3.2 Atomic Change Operations

An atomic change operation represents the most fundamental unit of architecture evolution that affects an individual architecture element. We provide the syntax for atomic change operations in Table 6.1 with a formal definition as:

Definition 6.1. Let OPR_{atomic} represent a collection of six atomic change operations that are expressed as: $OPR_{atomic} :=< Add_{por}, Rem_{por}, Mod_{por}, Add_{ept}, Rem_{ept}, Mod_{ept} >$ as addition (Add), removal (Rem) and modification (Mod) of ports (por) and endpoints (ept).

Change operations in Definition 6.1 are overloaded depending on the type of parameter (an endpoint or a port). An atomic change builds is fundamental to architecture change implementation.

ID	Operation	Syntax
1	Add_{por}	$ADD(por \in POR, cmp \in CMP)$
	- Add a new Port (*por*) to an existing Component (*cmp*).	
2	Rem_{por}	$REM(por \in POR, cmp \in CMP)$
	- Remove an existing Port (*por*) from an existing Component (*cmp*)	
3	Mod_{por}	$Mod(por \in POR, cmp \in CMP)$
	- Modify an existing Port (*por*) from an existing Component (*cmp*)	
4	Add_{ept}	$ADD(ept \in EPT, con \in CON)$
	- Add a new Binding (*bin*) to an existing Connector (*con*).	
5	Rem_{ept}	$REM(ept \in EPT, con \in CON)$
	- Remove an existing Binding (*bin*) from existing Connector (*con*)	
6	Mod_{ept}	$ADD(ept \in EPT, con \in CON)$
	- Modify an existing Endpoint (*ept*) in an existing Connector (*con*)	

Table 6.1: Syntax of Atomic Change Operations.

Syntax for Atomic Change

The syntax for atomic change operations is presented in Table 6.1, where OPR represents a given change operation applied to the given architecture element (parameter for change operation) and its cascaded impact on other elements. For example, in Table 6.1 the atomic operation with ID = 1 specifies an atomic change as: $Add(sendBill \in POR, PaymentType \in CMP)$.

The syntax above represents the addition of a port sendBill to an existing component PaymentType. We identified a total of six atomic change operations presented in Table 6.1. An atomic change enables a parameterised procedural abstraction that is fundamental to architecture change execution and provides the foundation for architectural change composition [Ducasse 2009].

6.3.3 Composite Change Operations

A composite change operation applies to composite architecture elements causing their evolution based on a set of pre-defined constraints that ensure architectural composition.

- *Syntax* for composite change operations is presented in 6.3 with a formal definition as below for addition (Add), removal (Rem) and modification (Mod) of components (cmp), connectors (con) and configurations (cfg).

- *Constraints* represent a set of predefined conditions (pre/post-conditions) that ensure the completeness of the construction of a composite architecture element from atomic ones. An explicit enforcement of constraints on the composite type changes distinguishes them from atomic ones. From an operational perspective, change ID 257 is identified as composite type change that enables the addition of a composite element by explicitly enforcing constraints - that combine atomic and composite type changes together - to maintain architectural consistency. The consequences of violating these constraints are orphan architecture elements (i.e.; component(s) without ports, connector(s) without endpoints).

Definition 6.2. Let $OPR_{composite}$ represents a collection of nine composite changes:

$$OPR_{composite} :=< Add_{cmp}, Rem_{cmp}, Mod_{cmp}, Add_{con}, Rem_{con}, Mod_{con}, Add_{cfg}, Rem_{cfg}, Mod_{cfg} >$$

Change operations in Definition 6.2 are overloaded depending on the type of parameter (component, connector or configurations). In order to analyse architectural changes, we query the change session graph in Figure 6.3 based on the type of change operation (OPR) and the correlation among its parameters (ARCH). An abstract syntax of the graph query is presented in Listing 6.1.

Listing 6.1: Abstract Syntax of Change Log Graph Query.

```
1  SET Start  ← 1
2  SET End ← Graph.Length()
3  SET ChangeID ← Start
4  SET Opr ← ""
5  SET Arch ← ""
6  While (ChangeID ← (ChangeID + 1) ≤ End)
7     Opr ← Graph.Node.getOPR()
8     Arch ← Graph.Node.getARCH()
9     If (Opr == "Add()" <OR> Opr == "Rem()" <OR> Opr == "Mod()")
10    If (Arch ==  "CMP.POR" <OR> Arch == "CON.EPT")
11 END While
```

The Listing 6.1 provides an abstract syntax for querying the change log graph. Instead of a concrete syntax of the query - expressing specific programming language statements to retrieve and manipulate the GraphML structure - we prefer an abstract and language independent representation in Listing 6.1. The abstract syntax helps us to hide the complexities of programming language-based representation focus and represent the necessary logic of the query that can be translated to some concrete syntax. The intent of this query is to retrieve the architectural changes containing only *Add()* or the *Remove()* or *Modify()* operation such that its parameter elements are co-related in the architectural model. The correlation among architecture elements is deïňĄned such that components and ports are co-related, connectors and endpoints are co-related. The main purpose of such a correlation is to enforce the structural integrity in terms of architectural composition with individual elements.

Therefore, in order to retrieve the specific change operations and architecture elements Line 1 to Line 5 in Listing 6.1 assigns specific. For example, the initial two lines represent the start and end (node) of the change log graph structure. Line 6 to Line 11 represents an iterative retrieval of each individual node in the change log graph (from start to the ending node). More specifically, in Line 07 and Line 08 the change operations and the associated architecture element from each node is retrieved. Finally, Line 09 and Line 10 ensures that all the change operations that support addition, removal or modification of the component ports and connector endpoints.

A partial result of the query is represented as the extracted sequence in Table 6.2. In Table 6.2 the sequence $ChangeID < 257, 258, 259 >$ represents the addition (257) of a new component PaymentType inside the configuration Payment. The addition of a component is followed by the addition (258, 259) of corresponding ports sendBill and payBill.

ChangeID	Operation	Architecture Element	Cascaded Impact
257	Add()	$PaymentType \in CMP$	$Payment \in CFG$
258	Add()	$sendBill \in POR$	$PaymentType \in CMP$
259	Add()	$payBill \in POR$	$PaymentType \in CMP$

Table 6.2: Retrieving Composite Changes (partial results of query in Listing 6.1).

Change composition also highlights the cascading impact of the change operationalisation that is propagated from top to bottom of the architectural hierarchy (configurations to components and components to ports). We identified a total of nine composite type changes as follows that are summarised in Table 6.3.

$$ADD(PaymentType \in CMP, Payment \in CFG) \prec ADD(sendBill \in POR, PaymentType \in CMP)$$

ID	Syntax	Operation
1	$ADD(cmp \in CMP, cfg \in CFG) \prec ADD(por \in POR, cmp \in CMP)$	Add_{cmp}
	- Add a new Component cmp with addition of a Port por.	
2	$REM(cmp \in CMP, cfg \in CFG) \prec REM(por \in POR, cmp \in CMP)$	Rem_{cmp}
	- Remove a Component cmp with removal of its Port por.	
3	$MOD(cmp \in CMP, cfg \in CFG) \prec MOD(por \in POR, cmp \in CMP)$	Mod_{cmp}
	- Modify a a Component cmp with modification of its Port por	
4	$ADD(con \in CON, cfg \in CFG) \prec ADD(ept \in EPT, con \in CON)$	Add_{con}
	- Add a new Connector con with addition of an Endpoint ept.	
5	$REM(con \in CON, cfg \in CFG) \prec ADD(ept \in EPT, con \in CON)$	Rem_{con}
	- Remove a Connector con with removal of its Endpoint ept.	
6	$MOD(con \in CON, cfg \in CFG) \prec ADD(ept \in EPT, con \in CON)$	Mod_{con}
	- Modify a Connector con with modification of its Endpoint ept.	
7	$ADD(O \times Add_{cmp} \prec P \times Add_{con})$	Add_{cfg}
	- Add a Configuration that contains O Components and P Connectors	
8	$REM(O \times Rem_{cmp} \prec P \times Rem_{con})$	Rem_{cfg}
	- Remove a Configuration containing O Components and P Connectors	
9	$MOD(O \times Mod_{cmp} \prec P \times Mod_{con})$	Mod_{cfg}
	- Modify a Configuration that contains O Components and P Connectors	

Table 6.3: A List of Composite Change Operations on Architecture Model.

Syntax for Composite Change

The syntactical representation for composite change is identical to the atomic change. However, a composite architectural change is applied to a composite architecture element. It requires a pre-defined sequencing of change operations to be followed based on pre-defined constrains detailed below. For example, in Table 6.3 the composite change operation with ID = 1 represents the addition of a new component (*cmp*) that must follow the addition of a port (*por*) as follows.

$$Add_{cmp} =: ADD(cmp \in CMP, cfg \in CFG) \prec ADD(por \in POR, cmp \in CMP)$$

Constraints on Composite Changes

In addition to the operational syntax, composite changes require preserving the operational constraints to ensure consistency of individual architectural elements and their composition relationships in the architecture. Composite change is specified as a set of constraints that are defined and expressed as below and also exemplified in the context of running example (cf. Figure 6.3):

Let a_i be the instance of an atomic elements that belong to a type A as $a_i \in A$ and c_x an instance of composite element C as $c_x \in C$ - where a_i contained-by c_x as: $a_i \triangleright c_x$. $\triangleright$ represents a contained-by relation and the constraints are:

$$CNS = \begin{cases} PRE: & \forall a_i \in A, \nexists a_i \triangleright C_x, \text{where } c_x \in C \\ POST: & \forall c_x \in C, \exists a_i \triangleright C_x, \text{where } a_i \in A \end{cases}$$

1. *Example - preconditions for component addition* - For example, to apply the addition operation, the port payBill must not already exist in the PaymentType component - in Figure 6.3, expressed as:

$$PRE: \forall payBill \in POR, \nexists payBill \triangleright PaymentType, \text{where } PaymentType \in CMP.$$

2. *Example - post-condition of component addition* For example, after change implementation the component PaymentType with a port payBill is successfully added:

$$POST: \forall PaymentType \in CMP, \exists payBill \triangleright PaymentType, \text{where } payBill \in POR.$$

The distinction between the atomic and composite change types have emerged from solutions for refactoring of object-oriented design/source-code. For example, the addition or removal of a class is a composite operation dependent on a number of individual changes that add or remove the attributes and operations to the newly added class [Lehnert 2012] based on the predefined constraints. In recent years, the distinction between atomic and composite change types is adopted from code refactoring solutions and utilised for architectural maintenance and evolution [Williams 2010]. Specifically, during architectural evolution the change type classification ensures that a correct composition hierarchy of the architecture model is preserved. To ensure the correct architectural composition there is a need to i) enable change composition (change execution), and ii) enforce constraints (preconditions and post-conditions) on composite changes both detailed below.

- **Composition of Architectural Changes -** We utilise the running example from Figure 6.3. For example, in Figure 6.3 the addition of a component PaymentType must follow the addition of two ports sendBill and payBill. This restriction is imposed by the composition hierarchy of component-based architecture model (cf. Chapter 2) and ensured by a set of constraints in terms of preconditions and post-conditions. The atomic ($\triangle A$) changes are combined with composite change ($\triangle C$) by enforcing the constraints (R is precondition and O is postcondition), $R \xrightarrow{\triangle A1, \triangle A2, ..., \triangle An} O$ is a transition from R to O.

$$\triangle C := R \xrightarrow{\triangle A1, \triangle A2, ..., \triangle An} O$$

  1. Composite change operations evolve the composite element (components and connectors) with atomic changes to reflect changes on ports of components and endpoints of operations.

  2. If a composite operation do not contain atomic changes it has three types of consequence on the architecture model as below.

     - *Orphan Component(s)* - refers to a component that cannot be interconnected to other components in the architecture model. More specifically, any architectural components that do not have at-least one port (either provider or requester) is unable to communicate to other components. Therefore, change composition must ensure that the composite architecture elements (e.g: component) contains the required sub-elements (e.g: port) to maintain the structural integrity of the architecture model.

     - *Orphan Connectors(s)* - refers to a connector that cannot interconnect two components in the architecture model. Any connector that do not have the endpoints (both source and target) cannot bind the provider and requester ports of the components.

     - *Comprising Structural Integrity* - when there are orphan components and connectors. For example, in order to avoid orphan components in the architecture model we must i) add at least one new port to the component, ii) move a port from another component to the newly added component, or iii) remove the component from the architecture model. The same applies to the orphaned connectors in terms of their endpoints.

6.4 Architecture Change Sequences

An architecture change sequence represents a sequential collection of change operations to perform higher-level change operations in terms of integration, composition, replacement etc. of components and connectors in architectural configurations

In Figure 6.4, we illustrate atomic changes and composite changes that can be combined as change sequences. For example, in Figure 6.4 an atomic change is limited to addition of a new port ($Por1$). The composite change enable architectural composition that specify addition of a new component ($Cmp1$) and its corresponding port ($Por1$). However, the composite change is limited to addition of co-related elements (cf. Listing 6.2) in architecture model. More specifically, a components and its port are co-related, similarly a connector and its binding are also cor-elated architecture elements. Therefore, composite operations could not be executed when we combine architectural changes that affect both the components and connectors. In such situations, we can exploit changes sequences that allow us to evolve the components and connectors. For example, in Figure 6.4 the change sequence represents the addition of two components ($Cmp1, Cmp2$) and their corresponding ports ($Por1, Por2$) that is followed by the addition of a connector $Con1$ to interconnect $Cmp1$, $Cmp2$.

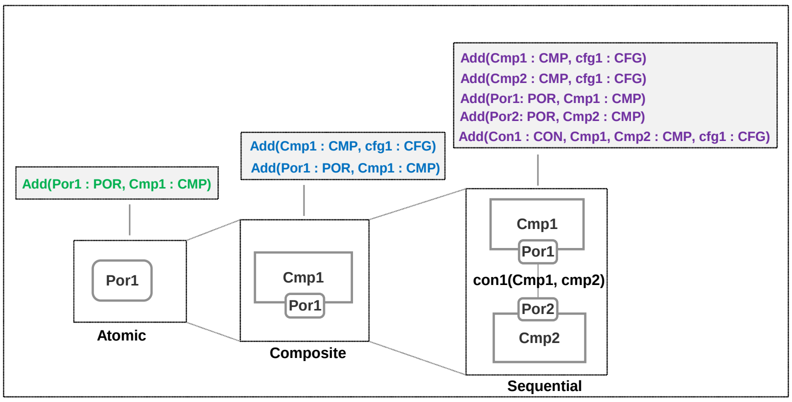

Add(Cmp1 : CMP, cfg1 : CFG)
Add(Cmp2 : CMP, cfg1 : CFG)
Add(Por1: POR, Cmp1 : CMP)
Add(Por2: POR, Cmp2 : CMP)
Add(Con1 : CON, Cmp1, Cmp2 : CMP, cfg1 : CFG)
Add(Cmp1 : CMP, cfg1 : CFG)
Add(Por1 : POR, Cmp1 : CMP)
Add(Por1 : POR, Cmp1 : CMP)
Por1
Atomic
Cmp1
Por1
Composite
Cmp1
Por1
con1(Cmp1, cmp2)
Por2
Cmp2
Sequential

Figure 6.4: A Summary of Syntactical Representation of Atomic Change Operations.

6.4.1 Order of Change Sequences

The order of change operations in a sequence can be applied and represented in the log in different ways [Zimmermann 2005]. In Figure 6.5, we present two distinct change sequences (S_1 and S_2) as extracted from change logs. The impact of change operations on the architecture model is presented on left while change representation in the log is represented on the right side in Figure

6.5. The intent for both these sequences is to introduce a new component in the architecture model and provide its interconnection to other components. However, a step-by-step comparison of both sequences based on matching the individual change operations in Figure 6.5 highlights that the user performed the identical changes but using a different order of change operations. First, we represent the intent of each change sequence and analyse the ordering of change operations.

- **Intent of Sequence (S_1)** is to integrate a new component PaymentType with its corresponding ports sendBill and payBill inside the configuration Payment. The newly added component is interconnected to CustPayment and BillerCRM components. This is represented as a sequence of six change operations (ChangeID 257 to 262).

- **Intent of Sequence (S_2)** is to add the component CustConsumption with ports billingData and payData in Billing configuration. The CustConsumption component provides details of the service consumption by the customer to CustBilling and CustPayment.

In Figure 6.5, a step by step match of the change operations reflects that the ordering of change operationalisation is irrelevant in sequential composition as long as the impact and definition of change remains same. When the intent and impact of change operations for S_1 and S_2 is identical, then we refer to this as the semantic equivalence ($\equiv$) among change sequences. However, if the order of change operations and their impact in S_1 and S_2 is distinct we refer to this as semantic non-equivalence ($\not\equiv$) of change instances.

Definition 6.3. Equivalent Change Sequences - Let $\equiv$ defines some equivalence relation of two change instances $S_j, S_k \in S : S_j \equiv S_k$ if and only if S_j and S_k are considered to be equivalent based on the following comparison properties:

1. **Property I - Type Comparison**: determines equivalence among two change operations in sequences S_j, S_k as $TypeCompare(Opr_m, Opr_n)$, where Opr_m, Opr_n are individual change operation in sequences S_j, S_k, respectively.

2. **Property II - Length Comparison**: performs length (total change operations) comparison among two sequences as $LengthCompare(S_j, S_k)$.

3. **Property III - Order Comparison**: determines the equivalence in ordering of change operations in two matching sequences $OrderCompare(S_j, S_k)$.

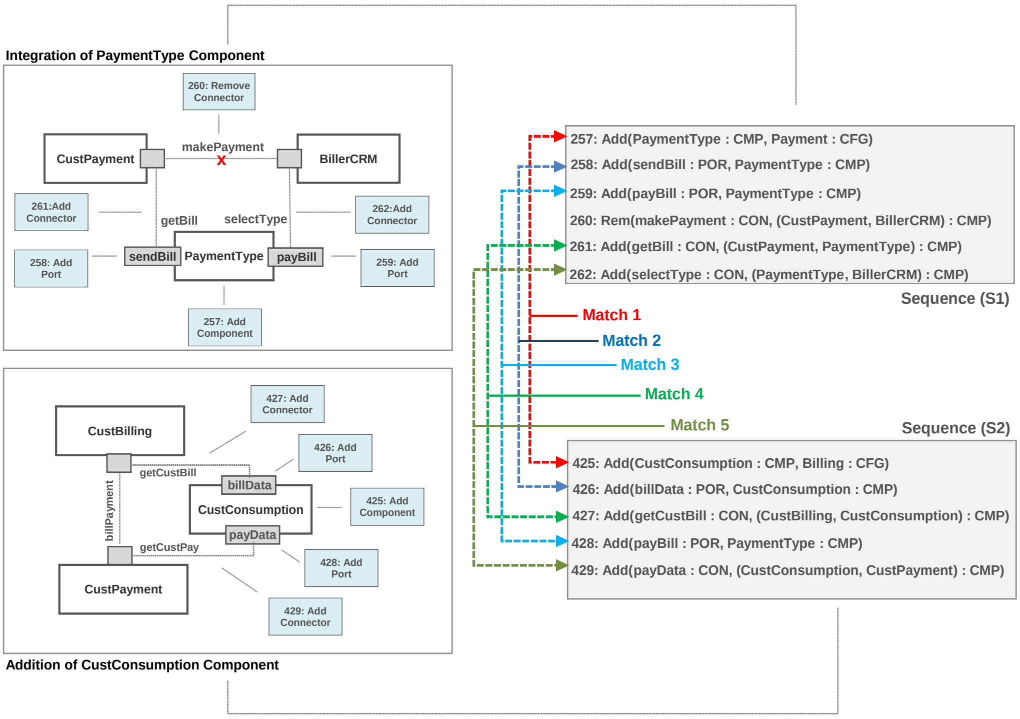

Integration of PaymentType Component
260: Remove Connector
CustPayment
makePayment
BillerCRM
261:Add Connector
getBill
selectType
262:Add Connector
258: Add Port
sendBill
PaymentType
payBill
259: Add Port
257: Add Component
CustBilling
427: Add Connector
426: Add Port
getCustBill
billData
billPayment
CustConsumption
425: Add Component
payData
getCustPay
428: Add Port
CustPayment
429: Add Connector
Addition of CustConsumption Component
257: Add(PaymentType : CMP, Payment : CFG)
258: Add(sendBill : POR, PaymentType : CMP)
259: Add(payBill : POR, PaymentType : CMP)
260: Rem(makePayment : CON, (CustPayment, BillerCRM) : CMP)
261: Add(getBill : CON, (CustPayment, PaymentType) : CMP)
262: Add(selectType : CON, (PaymentType, BillerCRM) : CMP)
Sequence (S1)
Match 1
Match 2
Match 3
Match 4
Match 5
Sequence (S2)
425: Add(CustConsumption : CMP, Billing : CFG)
426: Add(billData : POR, CustConsumption : CMP)
427: Add(getCustBill : CON, (CustBilling, CustConsumption) : CMP)
428: Add(payBill : POR, PaymentType : CMP)
429: Add(payData : CON, (CustConsumption, CustPayment) : CMP)

Figure 6.5: An Overview of the Operation Matching for Change Sequences.

Property I - Type Comparison (TypeCompare()) of Change Operations

It provides a comparison of the types of two change operations that is specified using a utility function $TypeComparison(OPR_x(arch_j \in ARCH), OPR_y(arch_k \in ARCH)) : returns < boolean >$. In Algorithm 6.1, OPR_x and OPR_y are compared (Line 03). Type equivalence depends on the type of change operation and the architecture element for a change operation to be categorised as type equivalent (return TRUE Line 04) or type distinct (returns FALSE Line 07). In Figure 6.6, we compare three change operations:

- **Type Equivalent Operations** the comparison of ChangeID (258, 259) returns TRUE as the type of change operation ($OPR.OprType : Add()$) and its parameter type ($OPR.ParamType : ARCH$).

- **Type Distinct Operations** the comparison of ChangeID (258, 257) returns FALSE as the type of the change operation (Add()) is identical, but operation parameter types are distinct (architecture elements).

We conclude that if both the change operation and its parameter among two change instances match we call the matching operations as type equivalent, or type distinct otherwise.

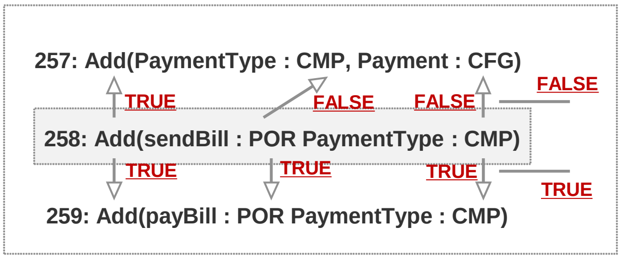

257: Add(PaymentType : CMP, Payment : CFG)
TRUE
FALSE
FALSE
FALSE
258: Add(sendBill : POR PaymentType : CMP)
TRUE
TRUE
TRUE
TRUE
259: Add(payBill : POR PaymentType : CMP)

Figure 6.6: Type Comparison of the Change Operations.

Algorithm 6.1 : $TypeCompare(OPR_x, OPR_y)$ to determine operational type equivalence

```
Input: Two Change Operations $OPR_x, OPR_y$
Output: Boolean $[TRUE/FALSE]$ to indicates if operations are type equivalent or type distinct
$OPR1\_Type \leftarrow OPR_x.Type \wedge OPR1\_Param \leftarrow OPR_x.Param$
$OPR2\_Type \leftarrow OPR_y.Type \wedge OPR2\_Param \leftarrow OPR_y.Param$
if $(OPR1\_Type \equiv OPR2\_Type \wedge OPR1\_Param \equiv OPR2\_Param)$ then
    return (TRUE)
end if
else
return (FALSE)
```

Property II - Length Comparison (LengthCompare()) of Change Sequences

It refers to comparing the length of two change sequences where length of a change sequence is defined by the number of change operation contained in it. It is given by the function $LengthCompare(S_x, S_y)$: $returns < integer >$ that is presented in Algorithm 6.2. First of all the length for both the sequences is identified individually (Line 02 - 04 and Line 05 - 07, respectively). Afterwards, the length equivalence of two change sequences S_x and S_y is determined by the following three conditions:

- **If, Sequence** S_x is equal in length to Sequence S_y represented as 0 that implies $S_x \equiv S_y$ (Line 08 - Line 10).
- **If, Sequence** S_x is smaller in Length to Sequence S_y represented as -N that implies $S_x < S_y$ by N operations (Line 11 - Line 13).
- **If, Sequence** S_x is greater in length to Sequence S_y represented as +N that implies $S_x > S_y$ by N operations (Line 14 - Line 16).

A numerical value (N) is returned as a result of the comparison (Line 17). For example, applying the $LengthCompare(S_1, S_2)$ function on Sequences S_1 and S_2 in Figure 6.5 returns 1, S_1 has six change operations compared to five operations in S_2.

Algorithm 6.2 : *LengthCompare*(S_x, S_y) to determine length equivalence among sequences

```
Input: Two Change Sequences $S_x, S_y$
Output: Number N that indicates variation among length of change sequences
$S_x.Length$, $S_y.Length \leftarrow 0$
while $S_x.Opr \neq$ NULL do
    $S_x.Length \leftarrow S_x.Length + 1$
end while
while $S_y.Opr \neq$ NULL do
    $S_y.Length \leftarrow S_y.Length + 1$
end while
if $S_x.Length \equiv S_y.Length$ then
    N $\leftarrow 0$
end if
if $S_x.Length < S_y.Length$ then
    N $\leftarrow S_x.Length - S_y.Length$
end if
if $S_x.Length > S_y.Length$ then
    N $\leftarrow S_x.Length - S_y.Length$
end if
return (N)
```

Property III - Order Comparison (Ordercompare()) of Change Sequences

It enables the comparison of the ordering of change operations of two sequences. It is given by the function $OrderCompare(S_x, S_y) : returns < boolean >$ as presented in Algorithm 6.3. It is normal for same user to perform similar changes using different sequencing of change operations also illustrated in Figure 6.5 (S_1, S_2). We distinguish between the following two types of sequences:

- **Exact Sequence**: Two given sequences are exact sub-sequences if they match on operational types, length equivalence and the ordering of the change operations.
- **Inexact Sequence**: Two given sequences are inexact matching sequences if their operational types and lengths are equivalent, but order of change operation varies.

First, we calculate the equivalence of the length of two sequences (Line 01), if the length are not equivalent further comparison is not performed (Line 03 - 05) - since the number of change operation varies. However,

- if the length and the order of change operation (calculated based on *TypeCompare*, Algorithm 6.1) iteratively matches (Line 06 - 08): a boolean value TRUE is returned (Line 10) to represent the exact sequences, otherwise:
- otherwise the length did matched but the order of change operations did not match (Line 11 - 12): a boolean value FALSE is returned

Algorithm 6.3 : $OrdCompar(S_x, S_y)$ to determine order equivalence among sequences

```
Input: Two Change Sequences $S_x, S_y$
Output: Boolean $[TRUE/FALSE]$ indicating if change sequences have a similar or distinct order
LengthEqu ← LenEquv($S_x, S_y$)
compareCount ← 0
if LengthEqu $\neq$ 0 then
    return (FALSE)
end if
while TypeCompare($S_{x_{OPR}}, S_{y_{OPR}}$) do
    compareCount ← compareCount + 1
end while
if compareCount $\equiv$ LengthEqu then
    return(TRUE)
else
    return (FALSE)
end if
```

Once the different types of architecture change sequences have been identified, the last step involves the classification of the different types of dependencies among change operations.

6.5 Dependencies of Change Operations

Abstracting atomic and composite change into a *sequence of change operations* (adjacent graph nodes) allow us to discover the types of dependencies that exist among change operations. We distinguish operational dependencies based on *commutative* and *dependent* change operations. Analysis of operational dependencies is vital to *investigate the extent to which architectural change operations are dependent or independent of each other*.

6.5.1 Commutative Change Operations

The concept of change commutativity is an effective mechanism to determine whether there exists a causal relation between consecutive change operations in a sequence. A causal relation refers to a sequential representation of change operations that can be applied in an arbitrary fashion without following a strict order of application (cf. Algorithm 6.3).

Operational Commutativity

We formally define the commutativity of architecture change operations as:

Definition 6.4. Let $arch_x$ be an instance of architecture model $ARCH$ and $S_x(OPR_i, OPR_j)$ and $S_y(OPR_j, OPR_i)$ be two change sequences, then the operations OPR_i, OPR_j in sequences S_x and

S_y are commutative if and only if:

$$OPR_i(arch_x) \prec OPR_j(arch_x) \Rightarrow arch'_x \wedge OPR_j(arch_x) \prec OPR_i(arch_x) \Rightarrow arch'_x.$$

The two change sequences S_x, S_y contain an equal number of change operations (cf. type and length comparison), while their order varies. The application of operations either in S_x or S_y on $arch_i$ results in an identical change to $arch'_i$

To illustrate the operational commutativity, we continue with change sequence S_1 in Figure 6.5, the component interconnection for PaymentType is achieved by following the sequence addition of component ports sendBill (changeID = 358) and payBill (changeID = 359) and connectors getBill (CustPayment, PaymentType) (changeID = 361) and selectType (PaymentType, BillerCRM) (changeID = 362). The above change sequence is represented as: ChangeID = (358, 359, 361, 362). However, the change sequence could also be applied using the sequences $ChangeID = \{(358, 361, 359, 362) or (359, 362, 358, 361)\}$, the impact of change remains exactly same. Such variations in operational ordering complicate the selection of a given sequence that executes a specific pre-defined change. Therefore, we utilise the concept of operational commutativity to determine if there exists a causal relation between consecutive change operations and the resultant impact of change.

6.5.2 Dependent Change Operations

If change operations are not commutative, we regard them as operationally dependent, i.e., the effect of the later change depends on its preceding change operation. Architecture change operations in a sequence are dependent if there exists a pre-defined order of application among change operations. Operational dependency is vital to preserve the compositional hierarchy of architecture elements - a component must be added before adding a port to it.

Operational Dependence

We formally define the dependency of architecture change operations as:

Definition 6.5. Let S_x and S_y be two change sequences applied on $arch_i \in ARCH$, we regards S_x and S_y as dependent if and only if S_x and S_y are non-commutative (following Definition6.4), and $S_x \prec S_y$ or $S_y \prec S_x$, either S_x precedes S_y or S_y precedes S_x in the order of their application (such an order cannot be altered).

For example, contrary to Definition 6.4, in Figure 6.5 the sequence S_1 of change operations must follow a predefined change sequence as an addition of a component PaymentType (ChangeID = 257), addition of a port sendBill (ChangeID = 258) and a connector getBill(PaymentType, CustPayment) (ChangeID = 261). More specifically, the addition of a component (257) must be followed by the addition of the corresponding port (258) and it is a connector (261) and this order of change operations cannot be altered.

6.6 Chapter Summary

In this chapter we focused on a taxonomical classification of architecture change operationalisation. A taxonomy of architecture change is essential to distinguish among the types of architecture change operations and dependencies that exist among operations. In addition, the definition of the syntax and composition of change operations enable a parametrise operational abstraction for change execution. In this and next chapter, we aim to answer **RQ 2** that highlights the needs for methods and techniques to discover evolutionary knowledge from change logs. This chapter provides a foundation to apply sub-graph mining [Agrawal 1995] - a formalised knowledge discovery technique - to discovery recurring change operationalisation and sequences that represent reusable, usage-determined architecture change patterns.

Chapter 7

Graph-based Discovery and Template-based Specification of Architecture Change Patterns

Contents

7.1 Chapter Overview

In this chapter, we focus on utilising the change log data - modelled as a sequential graph of architecture changes - to discover change patterns and specify them in a pattern template. We focus on *architecture change mining* that aims at exploiting repository mining concepts to investigate histories of architecture evolution for the discovery of architecture change patterns. As presented in Chapter 3 - acquisition of evolution reuse knowledge - the only notable work refers to the identification of architectural change patterns from object-oriented software [Tu 2002]. The fundamental distinction between [Tu 2002] and our solution is the level of discovery that represents a different software abstraction in terms of pattern discovery from source codes changes in [Tu 2002] and architecture evolution presented in this chapter.

Based on change log graphs (cf. Chapter 5) and a taxonomical classification of architecture change operationalisation (cf. Chapter 6), in this chapter we focus on the discovery of change patterns [Agrawal 1995] from sequential change log graph (cf. Chapter 5). In addition, we also provide a template-based specification of the discovered change patterns. We present pattern discovery as a continuous process for mining new patterns by investigating the history of architecture evolution over time. The newly discovered patterns represent the vocabulary of a pattern language that continuously evolves. In this chapter, we aim to answer **RQ 2** that focuses on methods and techniques to discover and represent evolution-reuse knowledge in terms of reusable change operationaisation and patterns. This chapter can be seen as a reference to establishing and enhancing architecture change mining (*pattern discovery*) as a complementary and integrated phase to architecture change execution (*pattern application*). We present a 4-step process for pattern discovery from change log graph in Figure 7.1 as below.

- **Step I -** The change log data is formalised as an attributed graph [Ehrig 2004] and represented using Graph Modelling Language (.GML) [Brandes 2002a] format - as a pre-processing to the pattern discovery process, detailed in Chapter 5.

- **Step II -** Based on investigating architectural changes and change taxonomy in Chapter 6, we derive a meta-model of pattern-based architecture evolution. This meta-model helps us to represent the structural composition of elements and their relationships to support pattern-based architecture evolution in Section 7.2.

- **Step III -** We present algorithms for graph-based pattern discovery [Agrawal 1995, H. Tong 2007] that include i) *Pattern Candidate Generation*, ii) *Pattern Candidate Validation* and iii) *Pattern Candidate Matching*. These algorithms are executed on change log graphs to discover architecture change patterns. Algorithmic details for pattern discovery are presented in Section 7.3.

- **Step IV -** We present discovered pattern instances and their specification in a change pattern

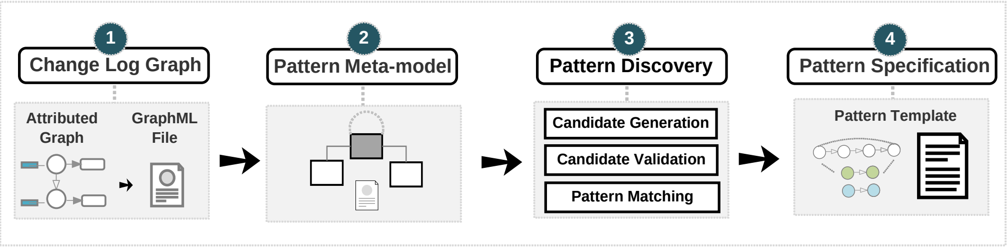


Figure 7.1: Process Overview for Log-based Change Pattern Discovery.

template [Harrison 2007] in Section 7.6.

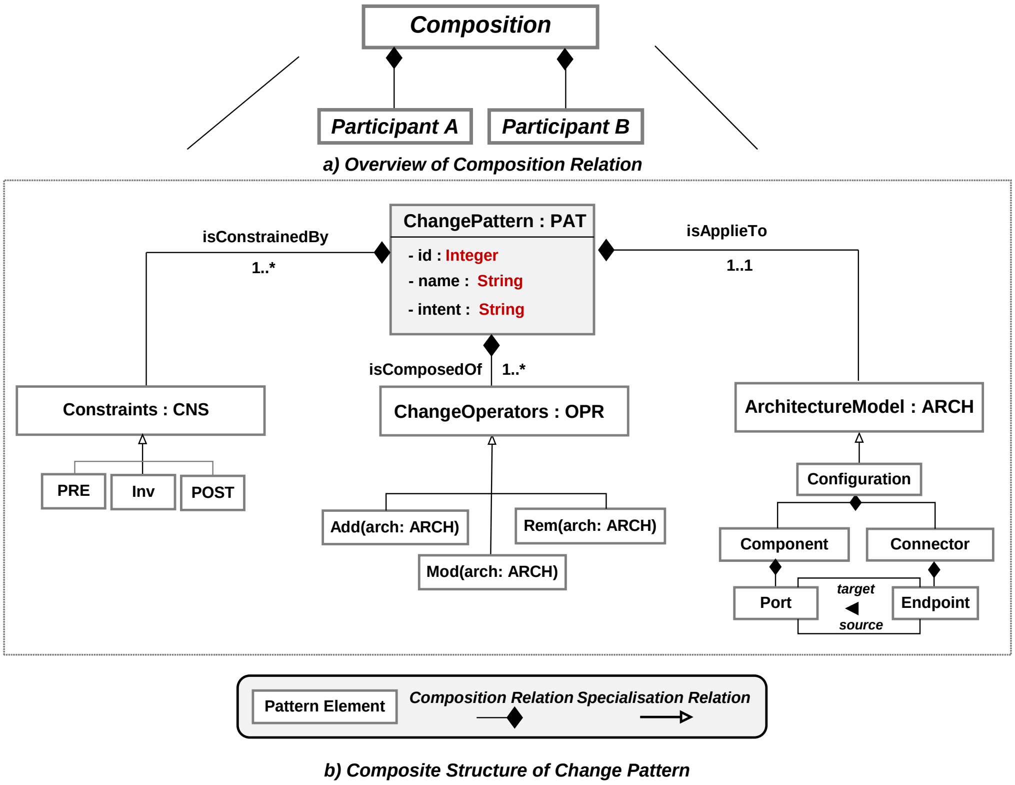
Patterns discovered in this chapter are fundamental to composition of a change pattern language. In the context of a pattern language discovered pattern instances and their possible variants represent a pattern language vocabulary [Zdun 2007, Porter 2005, Goedicke 2002] discussed in this chapter. Therefore, a pattern language vocabulary continuously evolves by discovering new change patterns from different logs over time.

7.2 A Meta-model of Pattern-based Architecture Evolution

In software architecture change logs, we observed that architectural changes can be operationalised and parametrised to support architecture evolution. More specifically, architecture elements that are added, removed, or modified are specified as parameters of change operations. The recurring architectural changes represent a change pattern as "*a generic, first class abstraction to support potentially reusable architectural change operationalisation*". A typical example of a change pattern is the replacement of a legacy component C1 with a new component C2 represented as Replace (C1, C2).

Composition of a Change Pattern - represents an abstraction of architectural changes as pattern composition, illustrated in Figure 7.2 a). Figure 7.2 a) represents the generalised structure for pattern *composition* while its concrete representation is illustrated in Figure 7.2 b) - using UML composition relations. For example, in Figure 7.2 a) at the top-level, the change pattern is a *composition* of various *participants* (change operations, constraints, architecture model) that are at a level below. The participants have their child also called *specialised elements*. This means to enable pattern-based evolution we need to specify architecture model (ARCH), change operations (OPR), constraints on architecture model (CNS) for composition of change patterns (PAT). The binary composition relationships among meta-model elements are expressed as: $\{isComposedOf,\ isAppliedTo,\ isConstrainedBy\}$.

For example, in Figure 7.2 the possible relation among a change pattern and change operators is expressed as: Pattern $\xleftarrow{isComposedOf}$ Operators, a change pattern is composed of change operations. Before discussing change pattern discovery, we discuss individual elements of pattern-based architecture evolution in the context of meta-model.



a) Overview of Composition Relation

b) Composite Structure of Change Pattern

Figure 7.2: Meta-model for Composition of Architecture Change Pattern.

7.2.1 Specifying the Architecture Model

To support architectural evolution, the descriptions for component-based software architectures (**CBSA**) [Medvidovic 1999, van der Aalst 2002] needs to be defined and constrained to achieve desired structure and semantics of source (before evolution) and target (after evolution) architecture models. We borrow architectural descriptions from Chapter 2 where we presented architecture meta-model as topological *configurations* (CFG) based on a set of architectural *components* (CMP) as the computational entities that are linked through *connectors* (CON) [Szyperski 2002]. Furthermore, architectural components are composed of component *ports* (POR), while connectors are composed of *endpoints* (EPT) to bind component ports. Both the source architecture and the target architecture must confirm to architectural meta-model as presented above and detailed in Chapter 2. The consistency of pattern-based change and structural integrity of architecture elements beyond component-based architecture model is undefined.

7.2.2 Specifying the Change Operations

Change operations represent a procedural abstraction to parametrise architectural changes that are fundamental to operationalising evolution. Our analysis of the change log goes beyond basic types in [Buckley 2005] to specify a set of *atomic* and *composite* operations (cf. Chapter 6) to enable structural evolution by adding (Add()), removing (Rem()) and modifying (Mod()) elements in architecture model. Atomic and composite change operators represent primitive changes [Barnes 2013] that are composed into pattern-based changes [Côté 2007] that abstract addition, removal and modification of components and connectors to facilitate frequent composition, decomposition, replacement type changes in an architecture model.

In the PatEvol framework, we specify change operations by means of graph transformation rules [Baresi 2002, Graaf 2007]. If architecture model is represented as a graph, graph transformation can be exploited to support its evolution.

7.2.3 Specifying Constraints on Architecture Model

To ensure that the structural integrity of individual architecture elements as well as the overall architecture model is preserved during and after architecture evolution, as set of constraints must be specified on the architecture model. We have presented constraints on architecture model as a set of preconditions (PRE), postconditions (POST) and architectural invariants in Chapter 5 (change log meta-model). Also, during the change operationalisation (Chapter 6) pre-conditions represented the context of architectural model before change execution. It represents complete or partial source architecture model that is evolved towards a target model. In addition, any change in the architectural structure must maintain the correctness of architectural invariant. Any violation of the architectural invariant results in an invalid instance of an architecture element. Finally, post-conditions represents evolved architecture model as a consequence of change operationalisation on architecture elements.

7.2.4 Specifying Change Patterns

We need to specify change pattern as a first-class abstraction that can be operationalised and parametrised to support potentially reusable architectural change execution defines as:

Definition 7.1. Change Pattern - Let a Pattern (PAT) represents a recurring, constrained (CNS) composition of change operationalisation (OPR) on architecture model (ARCH) expressed as:

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

A pattern enables a process-oriented approach to architecture change management describing the situation before and after the change (cf. Section 7.2.3 - constraints: PRE, INV, POST), along with the steps needed to implement the change (cf. Section 7.2.2 - change operations: Add(), Rem(), Mod()). The elements of the source architecture (cf. Section 7.2.1 - architecture model) $arch$ is evolved to $arch'$, where $arch, arch' \in ARCH$. In addition a pattern's *name* and its *intent* introduce pattern vocabulary. Pattern vocabulary provides an abstract view of problem-solution map (change operations, their impact on architecture model) captured by pattern name and intent.

In Chapter 2, we highlight pattern-based architecture evolution as a 3-step process including i) pattern identification, ii) pattern specification and iii) pattern instantiation. To support this, once change patterns are identified a collection or repository of pattern is required to support the i) specification (also pattern storage) and ii) instantiation (also pattern retrieval) steps. A pattern collection is essentially a repository infrastructure that facilitates an automated storage in terms of once-off specification and retrieval for multiple instantiation of discovered change patterns. This collection also supports pattern classification for a logical grouping of related patterns based on the types of architectural changes they support.

We also specify architecture change patterns in a change pattern template. We follow the guidelines in [Harrison 2007, Clements 2003] to develop an architecture change pattern template that provides a structured document to represent the name, intent to promote pattern as a solution that can be queried and retrieved. Template-based specification of architecture change patterns provide the foundation to derive sequencing among change patterns in a pattern language.

7.3 Algorithms for Change Pattern Discovery from Logs

Once change log data is formalised as an attributed graph [Ehrig 2004], the solution to pattern discovery problem lies with application of sub-graph mining approaches [H. Tong 2007] on change log graph. More specifically, our solution to graph-based pattern discovery lies with mining recurrent sequences (cf. Chapter 6) of change operations that is equivalent to discovering sub-graphs which occur frequently in a change log graph[1] G_{ACL}. In this section, we introduce the pattern

[1] Please note that the terminology *Change Log Graph* or *Log Graph* are used interchangeably that refer to a graph created from change log and is represented as G_{ACL}.

discovery problem as a modular solution and present pattern discovery algorithms. Pattern discovery algorithms[2] enable automation along with appropriate user intervention and customisation with algorithmic parametrisation of the pattern discovery process.

In Table 7.1, we provide a list of variables that facilitate parametrisation of algorithms for pattern discovery process. In Table 7.2, we outline a number of utility functions that are frequently used to maintain the modularity of the pattern discovery process in Figure 7.3.

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

Table 7.1: Parameters for Graph-based Pattern Discovery process.

Function(param)	**Return**	**Description**
$G_{ACL}.size()$	Integer	Get total number of nodes in log graph G_{ACL}
lookUp(P_C)	Boolean	Candidate P_C validation look-up in the invariant table
nodeMatching($n_j; n_k$)	Boolean	Bijective node matching based on $TypeEquv()$ (Section 5.1)
exactMatch($n_i; n_j$)	Boolean	Determine Exact match from candidate P_C to graph G_{ACL}
inexactMatch($n_i; n_j$)	Boolean	Determine Inexact match from candidate P_C graph G_{ACL}

Table 7.2: A List of Utility Methods for Pattern Discovery.

1. *Candidate Generation*: A pattern candidate (P_C) is a sequence of change operations in the change log graph (G_{ACL}) that represents a potential change pattern (PAT) depending on its occurrence frequency $Freq(P_C)$ in change log graph, such that $P_C, PAT \subseteq G_{ACL}$ illustrated in Figure 7.3a - Algorithm I.

2. *Candidate Validation*: The candidate validation is a significant step to eliminate the false positives, i.e., candidates that represent potential patterns but their application as change pattern may result in violating the invariants of the architecture model, illustrated in Figure 7.3b - Algorithm II.

3. *Pattern Matching*: Once the candidates are validated, we utilise graph matching (that compares graph nodes as change operations) to match recurring sequences and discover recurring sequences (P_C) as patterns (PAT), in Figure7.3 c - Algorithm III.

[2] The Java source code for pattern discovery is provided in **Appendix D**.

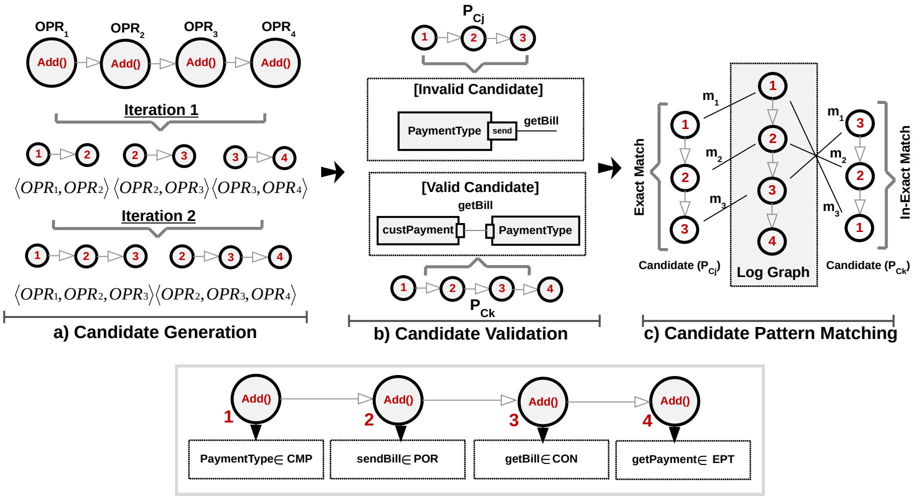


Figure 7.3: Overview of 3-Step Graph-based Pattern Discovery Process.

7.3.1 Algorithm I - Candidate Generation

As the initial step of the pattern discovery process, candidate generation aims at generating a set of pattern candidates P_C from an architecture change log graph G_{ACL}, as illustrated in Figure 7.3a. Each of the generated pattern candidate $p_{c_i} \in P_C$ represents a sub-graph of G_{ACL} as $P_C \subseteq G_{ACL}$. As presented in Table 7.1, the difference between a pattern candidate and a pattern is that the candidate must satisfy a specific occurrence frequency to be identified as a pattern. Therefore, a pattern candidate represents a change sequence (collection of graph nodes as change operations) as a potential pattern depending on its frequency $Freq(P_C)$ in G_{ACL}. We apply graph clustering approach [Brandes Ulrick 2007] on G_{ACL} to create graph clusters representing sub-graphs as pattern candidates in Figure 7.3a. Graph clusters from G_{ACL} are created based on the minimum and maximum length specified by the user as $minLen(P_C) \leq Len(P_C) \leq maxLen(P_C)$ as in Table 7.1. The size $Len(P_C)$ of a cluster (P_C) represents the total number of nodes in a cluster that ultimately represents the number of change operations in P_C. For example, in Figure 7.3a the user specifies $minLen(P_C) : 2$ and $maxLen(P_C) : 3$. In the first iteration candidates are generated such that the length of each candidate is two nodes with next iteration each candidate having three nodes. The generation of pattern candidates $PC_1, \ldots, PC_N$ (each representing an individual pattern candidate (P_C)) based on graph clustering [Brandes Ulrick 2007] is expressed as follows.

Specifically, the notation $G_{ACL(node\ +\ root)}$ (Line 7) represents an incremental generation of pattern candidates (Figure 7.3 a)), such that with each iteration the root element for each candidate is incremented by one node. This results in candidates having root as OPR_1 in first iteration and

OPR_2 in the next, until the maximum length of the candidates is reached (Line 6).

$$PatternCandidates = \left\{ \begin{array}{c} P_{C1} = \langle (OPR_1, OPR_2), (OPR_2, OPR_3), (OPR_3, OPR_4) \rangle \\ \\ P_{C2} = \langle (OPR_1, OPR_2, OPR_3), (OPR_2, OPR_3, OPR_4) \rangle \\ \\ P_{CN} = \langle (OPR_j, OPR_k, \ldots, OPR_n), (OPR_{j+1}, OPR_{k+1}, \ldots, OPR_{n+1}) \rangle \end{array} \right\} \tag{7.1}$$

1. **Input**: is a user specified change log graph G_{ACL} along with minimum $minLen(P_C)$ and maximum $maxLen(P_C)$ candidate lengths $minLen(P_C)$: 2 and $maxLen(P_C)$: 3 in Figure 7.3 a.

Algorithm 7.4 : candidateGeneration()

```
  Input: $G_{ACL}$, $minLen(P_C)$, $maxLen(P_C)$
  Output: $List(P_C)$
 1: buff($P_C$) ← $\phi$ {buffer to hold temporary candidates}
 2: root ← $G_{ACL}$.getRoot()
 3: for $candLength \leftarrow minLen(P_C)$ $candLength \leq maxLen(P_C)$ do
 4:    maxCandidates ← $G_{ACL.}$size() - candLength
 5: end for
 6: while root ≤ maxCandidates do
 7:    buff$(P_C)_{node}$ ← $G_{ACL(node\ +\ root)}$
 8:    node ← node + 1
 9:    root ← root + 1
10:    candLength ← candLength + 1
11: end while
12: List($P_C$) ← $\phi$ {List of final candidates}
13: for tempCand ← 0 tempCand ≤ buff($P_C$).Length() do
14:    if buff$(P_C)_{tempCand}$.Length() == buff$(P_C)_{Cand}$.Length() then
15:       if nodeMatching(tempCand, cand) == true and candidateValidation(cand) == true then
             List$(P_C)_{tempCand}$ ← buff$(P_C)_{cand}$
16:       end if
17:    end if
18: end for
19: return(List($P_C$))
```

2. **Process**: starts at the graph root with the selection of a single node and enumerating the temporary candidate list with adjacent node concatenation. Based on $minLen(P_C)$ and $maxLen(P_C)$, the temporary candidate list $buff(P_C)$ is generated as follows that is presented (Line 1 - 13): $buff(P_C) = \langle pc_1(OPR_1, OPR_2), pc_2(OPR_2, OPR_3), \ldots, pc_5(OPR_2, OPR_3, OPR_4) \rangle$. To avoid this exhaustive candidate list, the candidates in $buff(P_C)$ are iteratively matched to find specific candidates that occur at least more than once in G_{ACL}. We use the Breadth First Search (BFS) strategy [H. Tong 2007] over G_{ACL} with our matching function $nodeMatching(n_i; n_j)$

(Table 7.2) : $n_i.OPR \xrightarrow{match} n_j.OPR \wedge n_i.ARCH \xrightarrow{match} n_j.ARCH$ to generate the final candidate list: $List(P_C)$ (Line 14 - 19).

The breadth first strategy is applied to search for the candidates that occur more than once in the list of generated candidates distinguish with minimum and maximum lengths. For example, as illustrated in Figure 7.4 (a generalised view), during candidate generation, the candidates with minimum length (on left) are distinguished from candidates of maximum length (on right). With the BFS approach, (a) we analyse the individual candidates and (b) match it to its neighbouring candidates to see if a candidate occurs more than once. The search (Line 14) and matching process (Line 15) continues from candidates of minimum length to maximum length until all candidates are analysed (Line 13 - Line 18). The candidates that do not occur more than once are discarded to avoid an exhaustive generation of non-recurring candidates (Line 15 - Line 16).

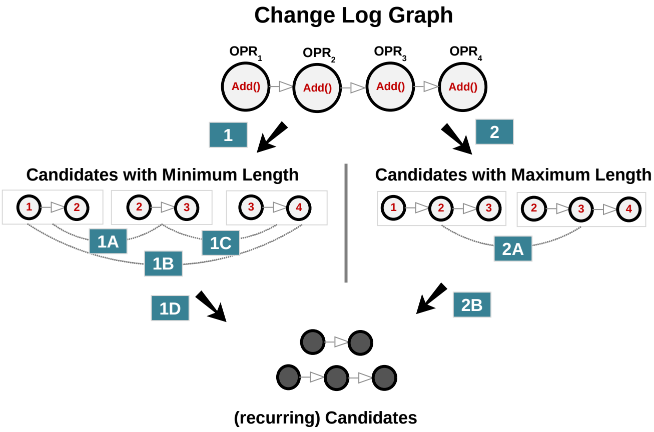


Figure 7.4: Overview of the Elimination of Non-recurring Pattern Candidates.

In addition, we ensure each candidate $pc_i \in List(P_C)$ is validated through candidateValidation(cp : G_{ACL}) (Line 18, detailed in Section 7.3.2).

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

7.3.2 Algorithm II - Candidate Validation

During candidate generation, there may exist some false positives in terms of candidates that violate the structural integrity (invariants) of the architecture model when identified and applied as patterns. For example, in Figure 7.3 b the candidate P_{C_j} represents three change operations as:

- *Operation I* adds a component PaymentType

- *Operation II* adds a port sendBill to component PaymentType
- *Operation III* adds a connector getBill

However, the connector does not provides interconnection among source and target ports (an orphan connector). Therefore, it is vital to eliminate the candidate pattern P_{C_j} that may violate architectural integrity (cf. 7.3b, invalid candidate). In contrast, the candidate P_{C_k} represents four change operations and provides interconnection among component ports in Figure 7.3b is referred to as a valid candidate. We eliminate invalid candidates through validation for each generated candidate *pc* against architectural composition (cf. Table 6.3) before pattern matching with algorithmic details below:

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

Algorithm 7.5 : candidateValidation()

```
    Input: $cp \in G_{ACL}$
    Output: $boolean[TRUE/FALSE]$ indicating if a candidate is valid of invalid.
 1: $isValid \leftarrow$ false
 2: iteration:
 3: for node $\leftarrow$ 0 node $\leq$ $pc$.Length do
 4:    if lookUp($pc.node.ARCH$) == true then
 5:       $isValid \leftarrow$ true
 6:    end if
 7:    if $isValid \leftarrow$ true then
 8:       break iteration
 9:    end if
10: end for
11: return($isValid$)
    {definition for lookup() function}
12: Boolean lookUp($pc.node.ARCH \in G_{ACL}$)
13: if $pc.node.ARCH \in CMP$ == true and $ARCH.\exists hasPOR(_{pro,req})$ == true then
14:    return(true)
15: end if
16: if $pc.node.ARCH \in CON$ == true and $ARCH.\exists hasEPT(_{src,trg})$ == true then
17:    return(true)
18: end if
19: return(false)
    {end lookup() function}
```

2. **Process**: includes look-up into the invariant table as Table 7.3 in terms of validating the architecture elements in the generated pattern candidates (in Line 3). More specifically it aims at detecting any orphaned components and connectors as a result of associated change operations. The orphaned component has no associated interconnection, while orphaned connectors have no associated component, indicated by Boolean value false.

   The definition for the invariant lookup() function is provided (Line 13 - Line 19). The architectural elements are confirmed to preserve the invariants (cf. Chapter 6): a) components containing provider or requester ports (where $ARCH.\exists hasPOR$ is an informal representation for a specific architectural element that must contain a port) and b) connectors containing source and target endpoint (for binding ports, where $ARCH.\exists hasEpt$ is an informal representation for a specific architectural element that must contain endpoints).

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

Element Type	Element Instance	Element Invariant
Component CMP	Component Instances ($cmp_i, \ldots, cmp_n$) $cmp_i, \ldots, cmp_n \in CMP$	$por_{pro} \vee por_{req} \in POR \triangleright cmp_i, \ldots, cmp_n \in CMP$
Port POR	Port Instances (por_{pro}, por_{req}) $por_{pro}, por_{req} \in POR$	
Connector CON	Connector Instances ($con_i, \ldots, con_n$) $con_i, \ldots, con_n \in CON$	$ept_{src} \vee ept_{trg} \in EPT \triangleright con_i, \ldots, con_n \in CON$
Endpoint EPT	Endpoint Instances (ept_{src}, ept_{trg}) $ept_{src}, ept_{trg} \in EPT$	

Table 7.3: Invariant Lookup Table

7.3.3 Algorithm III - Candidate Pattern Matching

Once the candidates are validated, the last step involves candidate pattern matching constrained by a user specified frequency threshold $Freq(P_C)$ for P_C in G_C. This means users specify that if a validated candidate in $List(P_C)$ occurs N times (determined by $Freq(P_C)$) a pattern PAT is discovered in change log graph G_{ACL}. We exploit sub-graph isomorphisms to match graph nodes (change operations) of P_C and G_{ACL} iteratively.

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

2. **Process**: includes retrieving each candidate from $List(P_C)$ and finds its exact or possible inexact instance in G_{ACL}. In any given match from P_C to G_{ACL} the number of nodes must be equal (Line 13).

   We exploit the change sequence properties (cf. Chapter 6) to specify: if and only if all the nodes in the candidate match the corresponding nodes in change log graph we refer to this as P_C is isomorphic to G_{ACL} as: $nodeMatching(P_C, G_{ACL})$ =

$$\left\{ \begin{matrix} \langle P_{C_1}(OPR_1, OPR_2) \rangle & \cdots & \langle P_{C_n}(OPR_i, OPR_j, \ldots, OPR_k) \rangle \\ \vdots & \ddots & \vdots \\ G_{ACL}(OPR_1, OPR_2, \ldots, OPR_N) & & G_{ACL}(OPR_1, OPR_2, \ldots, OPR_N) \end{matrix} \right\} \tag{7.2}$$

Algorithm 7.6 : patternMatch()

```
Input: $List(P_C)$, $Freq(P_C)$, $G_{ACL}$
Output: $pList(PAT, Freq(PAT))$
1: gCand(pc : $G_{ACL}$) ← $\phi$ {hold extracted nodes from $G_{ACL}$}
2: root ← $G_{ACL}$.getRoot()
3: for cand ← 0 cand ≤ $List(P_C)$. Length do
4:    freq ← 0 {to count frequency of $P_C$ in $G_{ACL}$}
5: end for
6: while root ≤ $G_{ACL}$.getLeaf() do
7:    exactMatch ← 0
8:    inexactMatch ← 0
9: end while
10: for node ← root node ≤ $List(P_C)_{cand}$ .Length() do
11:    gCand(node ← node + 1) ← (root ← root + 1)
12: end for
13: if List($P_C)_{cand}$.Length() == gCand(root).Length() then
14:    for node ← root node ≤ List($P_C)_{cand}$ .Length() do
15:       if match(List($P_C)_{cand}$.node, gCand$(root).node$ == true) then
16:          exactMatch ← exactMatch + 1
17:       end if
18:       if inexactmatch($List(P_C)_{cand}.node$, gCand$(root).node$ == true) then
19:          inexactMatch ← inexactMatch + 1
20:       end if
21:    end for
22: end if
23: if exactMatch == List($P_C)_{cand}$.Length() OR inexactMatch == List($P_C)_{cand}$.Length() then
24:    freq++
25: end if
26: if freq ≥ $Freq(P_C)$ then
27:    pList(PAT, Freq(PAT)) ← (List($P_C)_{cand}$, freq)
28: end if
```

- *Exact Match* : It is based on exact and *partial exact* sequences in **Chapter 6**. An exact match requires that there must exist a bijective mapping among types of change operator and the type of architecture element in attributed nodes that is given as a utility function (cf. Table 7.2) exactMatch($nodeMatching(n_i; n_j))[\forall(i, j) = 1 \ldots N]$ (cf.) that utilises the function (cf. Table 7.2) $nodeMatching(n_i, n_j)$ method it enables finding an exact match among the candidate nodes P_C (node) to the corresponding nodes in the change log graph G_{ACL} (node) in Figure 7.3 c. In addition the ordering of matching nodes from $List(P_C)$ to G_{ACL} must be same (exact change sequences already explained in Chapter 6). If such an exact instance is found, the candidate's frequency is incremented and matching is repeated (Line 15, 16), otherwise:

- *Inexact Match* : It is based on exact and *in-exact* and *partial in-exact* sequences in **Chapter 6**. The order of matching nodes from $List(P_C)$ to G_{ACL} is not always same. In this case, $inexactMatch(nodeMapping(n_i; n_j))[\forall(i \rightarrow j) = 1 \ldots N]$ that utilises the $nodeMatching(n_i, n_j)$ method to find an inexact match among the candidate nodes P_C (node) to the corresponding nodes in the change log graph G_{ACL}(node) in Figure 7.3 c. The candidate's frequency is incremented and matching is repeated until leaf node (Line 18, 19).

3. **Output**: is a list of identified patterns consisting of the pattern instance PAT and its corresponding frequency $Freq(P_C)$. A given candidate is an identified pattern (exact or inexact) if its frequency is greater or equal to specified threshold: $freq(PAT) \geq Freq(P_C)$.

7.4 Complexity of Change Pattern Discovery

After detailing pattern discovery algorithms, we also highlight some of the algorithmic efficiency concerns that results in a significant and often exponential increase of computation time for pattern discovery. In this section, we highlight the possible improvements (as a tradeoff between the Accuracy vs efficiency) of the pattern discovery algorithms - also pinpointing the relative benefits and limitation of the proposed pattern discovery techniques. Accuracy refers to an accurate discovery of the existing patterns such that no existing patterns in the log may be skipped or remains undiscovered, whereas efficiency refers to the time taken to discover the existing patterns. In an ideal situation, we want to discover all or the maximum of the available patterns (high accuracy) in the shortest time (increased efficiency). We evaluate the accuracy and efficiency of the pattern discovery process in detail later in the thesis.

Specifically, minimising the complexity of the frequent sub-graph mining (FSM) approach (that exploits sub-graph isomorphism) is fundamental to the efficiency of pattern discovery process. By means of sub-graph isomorphism, the nodes of sub-graph(s) (a.k.a. candidates) are iteratively matched to the nodes of a log graph to discover recurring sub-graphs (a.k.a. patterns). However, sub-graph isomorphism is a complex problem that is known to be NP-complete [Conte 2004], a set of problems not known whether these could be solved in polynomial time. For example, considering Figure 7.3, step 1; central to the operation of FSM for pattern discovery is the generation of a number of candidates, where the number of generated candidates is proportional to the size of the log graph (cf. Figure 7.3) and as the size increases, the total time for matching graphs for pattern discovery increases. Relating to this, we have two challenges:

- *Generation of potentially large number of pattern candidates* - as illustrated in Figure 7.3, even with a very small graph size (N = 4) and limited number of candidate lengths (min, I = 2, max J = 3), a total of five distinct candidates are generated based on a generalised relation: $TotalCandidates(I, J) := (N - I + 1) + (N - J + 1) + \ldots + (N - N + 0)$. However, generating the candidates does not involve matching their corresponding sub-graph with a log graph. Instead, each candidate is generated simply by appending adjacent nodes iteratively (cf. Algorithm 7.4, Line 1 - 13).

- *Matching of potentially large number of candidate graphs with log graph* - as the total number of candidates are matched each at once. Also as their length grows by one (based on user-specfied I and J lengths), this can lead to a potentially large number of sub-graphs to be matched, leading to the complexity of pattern mining [Conte 2004] - for NP-complete cases. This means that for a graph having a total of 2500 nodes ($N = 2500, I = 2, J = 3$) a total of approximately 5000 candidates (sub-graphs are generated). This represents a worst-case scenario as the smaller the value of I and J, more candidates will be generated. The solution starts to become impractical as the size of N grows, unless some reduction in the total number of candidates (sub-graphs to be matched) is performed. Therefore, prior to graph matching for pattern discovery (cf. Algorithm 7.6), the possible reductions of candidates to be matched is performed as follows.

The exhaustive list of generated candidates (Algorithm 7.4, Line 1 - 13, Figure 7.3) is refined by removing the non-recurring candidates (Algorithm 7.4, Line 14 - 19). Specifically, if there is no recurring sequence; we do not have any pattern in the graph (extreme and nonexistent case in the context of pattern discovery precision). This means that i) either a candidate is eliminated if not recurring and from the remaining ii) if a candidate occurs more than once its duplicate is removed. To illustrate, we extend the original graph (G_{org}) in Figure 7.3 that represents a sequential collection of four change operations expressed as $G_{org} := \{OPR_1; OPR_2; OPR_3; OPR_4\}$. Please note that ($OPR_1 - OPR_4$) are simplistic representation of their corresponding change operations and (;) is a sequence among operations. The intent of the artificially generated extended graph (G_{ext}) is to represent the recurring changes in the original graph expressed as $G_{org} := \{OPR_1; OPR_2; OPR_3; OPR_4; OPR_1; OPR_2; OPR_3; OPR_4\}$. Now, when we generate the pattern candidates from G_{org} with:

- *minimum candidate length* (I) is 2, we have a total of 7 generated candidates: $Cand_1$:= $\{OPR_1; OPR_2\}$ $Cand_2 := \{OPR_2; OPR_3\}$ $Cand_3 := \{OPR_3; OPR_4\} Cand_4 := \{OPR_4; OPR_1\}$ $Cand_5 :=$ $\{OPR_1; OPR_2\}$ $Cand_6 := \{OPR_2; OPR_3\}$ $Cand_7 := \{OPR_3; OPR_4\}$
- *maximum candidate length* (J) is 3, we have a total of 6 generated candidates: $Cand_8$:= $\{OPR_1; OPR_2; OPR_3\}$ $Cand_9 := \{OPR_2; OPR_3; OPR_4\}$ $Cand10 := \{OPR_3; OPR_4; OPR_1\}$ $Cand_{11} :=$ $OPR_4; OPR_1; OPR_2\}$ $Cand_{12} := \{OPR_1; OPR_2; OPR_3\}$ $Cand_{13} := \{OPR_2; OPR_3; OPR_4\}$.

We find all the candidates that occur at least once that are such as $Cand_1$, $Cand_5$ or $Cand_8$ and $Cand_{12}$, while the non-recurring candidates such as $Cand_4$ and $Cand_{10}$ are removed (Algorithm 7.4, Line 14 - 19). The recurring candidates are counted only once as $Cand_1$ and $Cand_4$ represents exactly same instance of change operation. Therefore, the candidates are reduced to 5 $Cand_1$, $Cand_2$, $Cand_3$, $Cand_8$ and $Cand_9$. However, based on the varying length of the graph and the values of the minimium (I) and maximum (J) length of the candidates the candidate reduction is unpredictable.

7.4.1 Performance Trade-offs - Accuracy vs Efficiency of Pattern Discovery

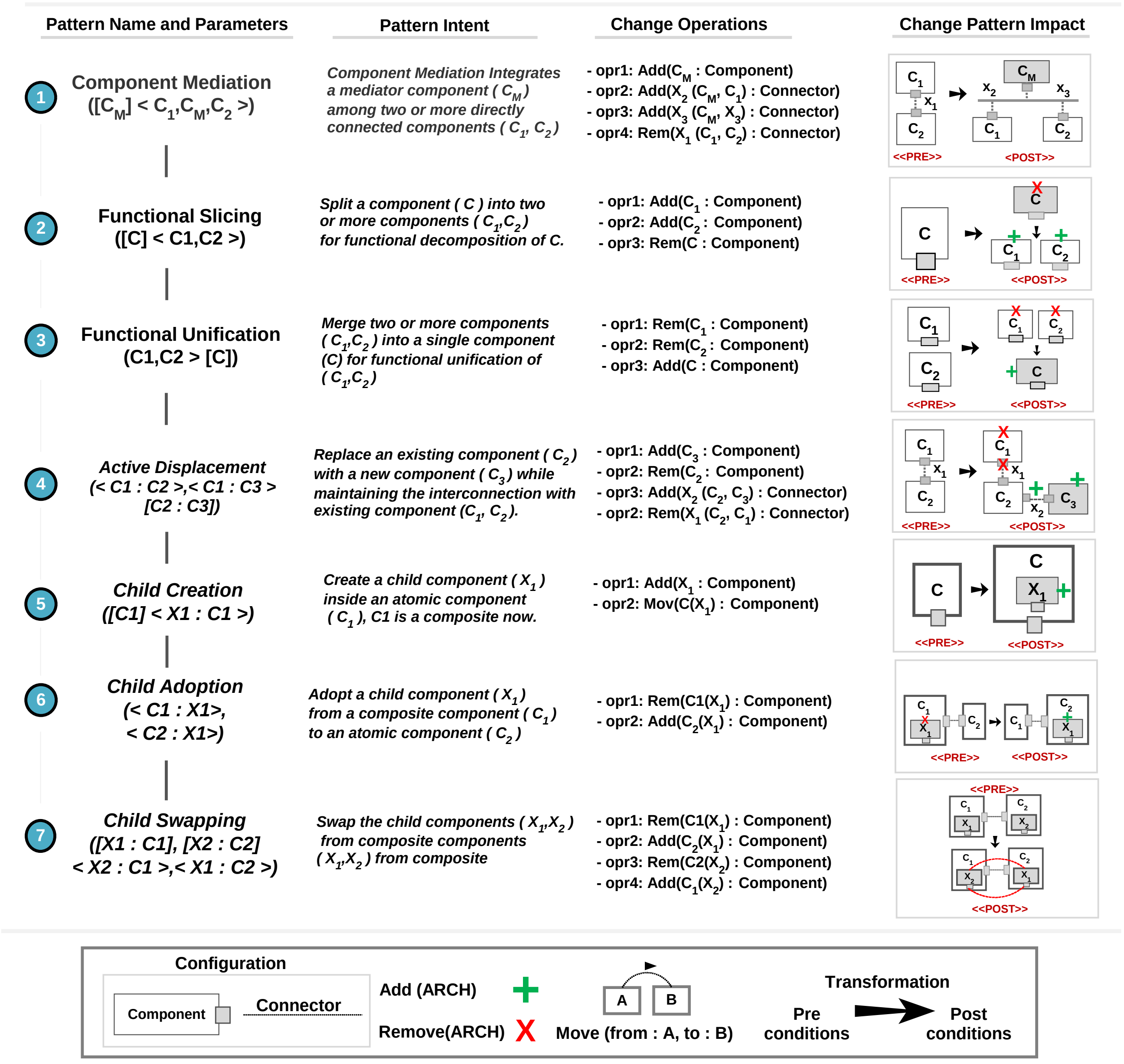
In the scenario above, the reduction technique (elimination of non-recurring candidates) reduces the search space. However, considering a graph of N nodes there still is a large number of subgraphs that need to be matched. In such case, we need to apply a trade-off between accuracy and efficiency of the pattern discovery process. Specifically, we select a complete solution that ensures a high accuracy of pattern discovery while compromising the algorithmic runtime efficiency. Alternatively, we could derive an approximate solution - specifying higher values for minimum and maximum lengths of pattern candidates - to minimize the pattern candidates while enhancing the algorithmic runtime efficiency. A known method for such an approximation is heuristic as a technique to solve problems more quickly by finding an approximate solution when classic methods fail to find an efficient solution [Desrosiers 2011]. The typical examples for heuristic-based and graph supported pattern mining are [Ketkar 2005, Ghazizadeh 2002]. Specifically, the approach named Subdue [Ketkar 2005] focuses on compressing the potential patterns by producing a fewer number of highly interesting patterns than to generate a large number of patterns from which interesting patterns need to be identified. The results of Subdue suggest that the solution can efficiently discover frequent patterns which are fewer in number but can be of higher interest - approximating the number of pattern candidates. Another example is the algorithm called SEuS [Ghazizadeh 2002] that uses a data structure called summary to construct a lossy compressed representation of the input graph. The authors indicate, this summary data-structure is useful only when the input graph contains a relatively small number of frequent sub-graphs with high frequency, and is not effective if there are a large number of frequent sub-graphs with low frequency.

We choose a complete solution rather than heuristic-based approach (selecting higher values for the minimum and maximum lengths of candidates) as we have a slightly smaller log graph and we prefer accuracy over performance. A survey of solutions for graph-based pattern discovery [Jiang 2012] and sequential pattern mining solution [Agrawal 1995] highlights that parametrisation is also vital in the reduction process. More specifically, based on parameters (cf. Table 7.1) if the values for minimum and maximum candidate lengths, pattern frequency threshold are high there are few matches to be performed (lesser candidate generation). However, by selecting higher values for minimum and maximum candidate length the risk lies with skipping the potential patterns that have smaller candidate lengths and such a solution is not preferred in this thesis. A practical demonstration is Figure 7.3 - step 2, where if the user can specify the minimum length more than 5 a significant number of candidates can be minimised (since 4 or more operations are required for a candidate to be valid). However, in such a case we may miss out the patterns that have a length of less than 2.

In some other cases (nested graphs vs sequential graphs) there is a need to validate the solution with potentially ultra-large (often artiïňĄcial) data sets for evaluation purposes that is not validated in our case. Also, the evolution histories of software [Kagdi 2007] are comparatively smaller and less complex when compared to mining chemical structures [Conte 2004] and business-oriented data [Agrawal 1995].

7.5 Discovered Change Patterns from Logs

We provide a listing of discovered pattern - as a results of executing the pattern discovery algorithms on architecture change log graphs - in Figure 7.5. In Figure 7.5, we only provide a pattern overview in terms of the following elements.



	Pattern Name and Parameters	Pattern Intent	Change Operations	Change Pattern Impact
1	Component Mediation ($[C_M] < C_1, C_M, C_2 >$)	*Component Mediation Integrates a mediator component (C_M) among two or more directly connected components (C_1, C_2)*	- opr1: Add(C_M : Component) - opr2: Add(X_2 (C_M, C_1) : Connector) - opr3: Add(X_3 (C_M, X_3) : Connector) - opr4: Rem(X_1 (C_1, C_2) : Connector)	
2	Functional Slicing ([C] < C1,C2 >)	*Split a component (C) into two or more components (C_1,C_2) for functional decomposition of C.*	- opr1: Add(C_1 : Component) - opr2: Add(C_2 : Component) - opr3: Rem(C : Component)	
3	Functional Unification (C1,C2 > [C])	*Merge two or more components (C_1,C_2) into a single component (C) for functional unification of (C_1,C_2)*	- opr1: Rem(C_1 : Component) - opr2: Rem(C_2 : Component) - opr3: Add(C : Component)	
4	*Active Displacement* *(< C1 : C2 >,< C1 : C3 > [C2 : C3])*	*Replace an existing component (C_2) with a new component (C_3) while maintaining the interconnection with existing component (C_1, C_2).*	- opr1: Add(C_3 : Component) - opr2: Rem(C_2 : Component) - opr3: Add(X_2 (C_2, C_3) : Connector) - opr2: Rem(X_1 (C_2, C_1) : Connector)	
5	*Child Creation* *([C1] < X1 : C1 >)*	*Create a child component (X_1) inside an atomic component (C_1), C1 is a composite now.*	- opr1: Add(X_1 : Component) - opr2: Mov(C(X_1) : Component)	
6	*Child Adoption* *(< C1 : X1>, < C2 : X1>)*	*Adopt a child component (X_1) from a composite component (C_1) to an atomic component (C_2)*	- opr1: Rem(C1(X_1) : Component) - opr2: Add(C_2(X_1) : Component)	
7	*Child Swapping* *([X1 : C1], [X2 : C2] < X2 : C1 >,< X1 : C2 >)*	*Swap the child components (X_1,X_2) from composite components (X_1,X_2) from composite*	- opr1: Rem(C1(X_1) : Component) - opr2: Add(C_2(X_1) : Component) - opr3: Rem(C2(X_2) : Component) - opr4: Add(C_1(X_2) : Component)	

Figure 7.5: List of Discovered Architecture Change Patterns.

1. **Pattern Name and Parameters**[3] - Name provides an identification of a pattern to its user. Parameters represent the affected architecture elements based on pattern application.

2. **Pattern Intent** [4] Represents a high-level pattern description in terms of the objective of pattern usage. In Figure 7.5 the *Pattern Name* Component Mediation specifies the intent as a pattern that enables the integration of a mediator component C_M with componentS C_1, C_2.

3. **Change Operationalisation** It provides an operational execution of architectural changes as a constrained composition of operators to enable architecture evolution.

4. **Pattern-based Change Impact** It represents the impact of change pattern on architecture model represented as the pre-conditions and post-conditions of change pattern.

The details of the prototype for pattern discovery (GPride - Graph-based Pattern Identification) are provided in **Appendix D**.

7.5.1 Discovering and Generalising the Pre/Post-conditions of Change Patterns

The preconditions and post-conditions as an integral part of the pattern are being mined as part of the pattern discovery process. During the pattern mining process, the constraints represent the essential conditions that must be preserved or satisfied by the given pattern. These constraints are the restrictions or conditions imposed by the domain to which the pattern is applied. For example, in [Pei 2002] the pattern mining from a database of medical examination, a pattern named body fever is valid only if it has constraints such as body temperature (in the range of 100 to 106 Celsius) with headache or cough (lasting on average 4 hours). In this case the domain (i.e.; medical examination) specify the constraints in terms of specific body temperature, headache and cough as necessary conditions associated to patterns of body fever. In contrast, with the architecture change patterns the constraints are represented as preconditions (conditions before the application of pattern) and post-conditions (conditions after the application of pattern). For example, in Figure 7.6 (an instance of *ComponentMediation* pattern from Figure 7.5) the constraints are associated to the domain of change patterns, i.e.; component-based software architectures and ensure the structural integrity of architecture model in terms of the required components and connectors.

[3] Please note that Pattern Name is vital to maintain the identity of a pattern in a collection [Buschmann 2007]. The selection of an appropriate name for a pattern is subjective to the choice of pattern author or users of a patterns.

[4] The term Intent was first used in the GoF book to describe the primary objective of a pattern. However, now a days it is also common among pattern authors to reflect to it as pattern overview - pattern thumbnails or problem/solution-pairs.

It is vital to mention that the constraints are implicitly represented in the change log and are associated to change operations, i.e., the addition or removal of the architectural elements (change operations) results in architecture model before and after pattern application (pre/post-conditions) as in Figure 7.6. More specifically, in the context of Figure 7.6 *ComponentMediation* pattern the architectural elements *PaymentType* as component and *selectType*, *custPayment* as connectors are being added (i.e., *Add()* operation) as part of post-conditions, while the architectural elements that are removed makePayment as connector (i.e., *Rem()* operation) represent the part of preconditions.

The pattern discovery and specification prototype (described in **Appendix D**) helps with discovery of constraints and then visualise the constraints as conditions before and after pattern application representing the Change Pattern Impact on the architecture model. We illustrate the discovery and generalisation of the preconditions and post-conditions as a two-step process with the help of *ComponentMediation* below.

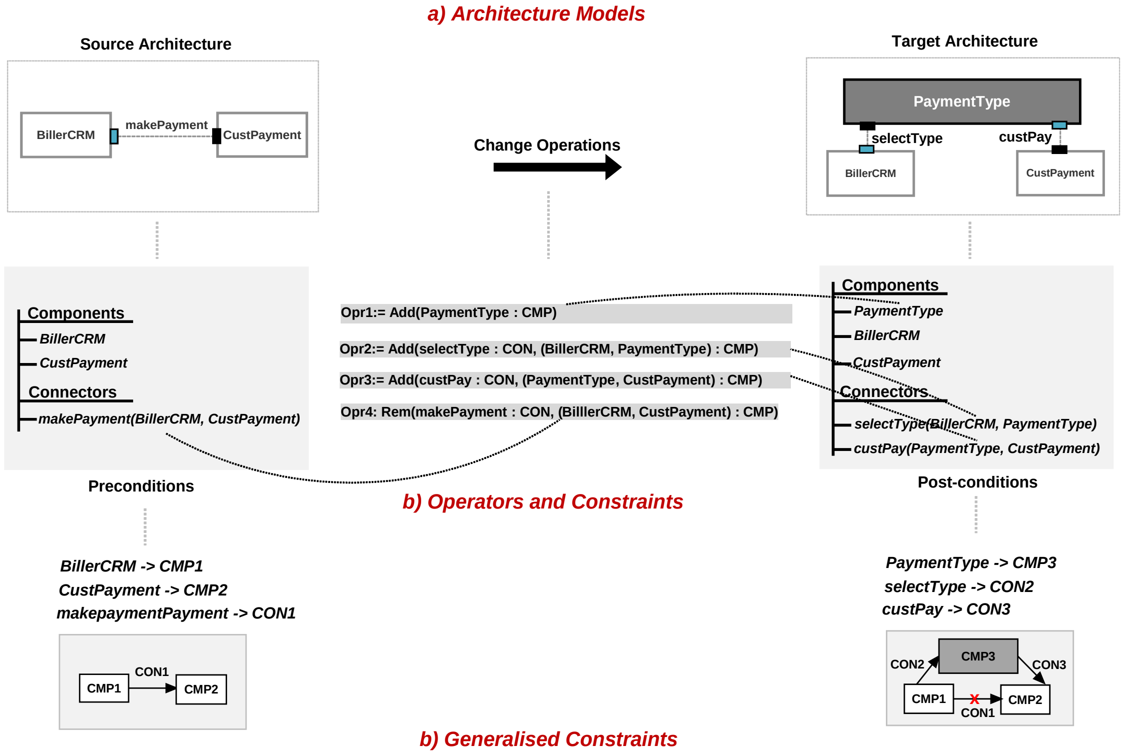


Figure 7.6: An Overview of Pattern Constraints Discovery and Generalisation Process.

- **Discovery of the Pattern Preconditions** - the preconditions are simply the architecture elements in the architecture model before any change operation is applied. For example, in

Figure 7.6 the source architecture model represents a connector *makePayment* with existing components *BillerCRM* and *CustPayment*. Any element that is being removed from the architecture model is a part of the preconditions.

- **Discovery of the Pattern Post-conditions** - the post-conditions represent the architecture elements in the architecture model after any change operation is applied. For example, in Figure 7.6 *Opr1* to *Opr3* represent the addition of components that were not part of the architecture model and therefore part of the evolved architecture.

- **Generalisation of the Pre/Post-conditions** - once the pre/post-conditions are specified, the next step involves generalising the constraints that refers to a generic representation of the names of the architectural elements. These generic names are part of pattern specification that abstracts the specific names and instances of architecture elements with more generic representation. In this context, the names of architecture models are simply replaced with a more appropriate general name. For example in Figure 7.6, the name of the specific instance of a component *BillerCRM* is replaced with CMP1 and alternatively the connector *makePayment* is renamed to *CON1*. In case of a concrete instance of a pattern the process can be reversed, i.e., to replace the generic names with more specific names of the concrete architecture elements.

7.6 Template-based Specification of Architecture Change Patterns

A change pattern template provides a structured documentation of individual patterns. We provide a formal template for pattern specification that is based on the meta-model for pattern-based evolution (Figure 7.2) and the guidelines for documenting patterns and styles presented in [Harrison 2007, Clements 2003]. As outlined in Chapter 2, we follow a 3-step process to facilitate *acquisition* and *application* of change patterns as i) pattern identification (i.e., pattern discovery), ii) pattern specification (i.e., pattern documentation) and iii) pattern instantiation (i.e., pattern application). Therefore to support the reuse of discovered patterns, a template-based specification for change patterns is provided. We map each of the elements of pattern meta-model (from Figure 7.2) to the possible relationships among the pattern elements. An overview of the pattern template is provided in Table 7.4, we utilised the graph modelling language (.GML) [Brandes 2002a] for graph-based representation of pattern template. We already explained (cf. Chapter 2) the rationale behind graph-based specification of architecture change patterns and provide a quick review about our decision as below. We use the syntax:

(PatternElement) <ElementAttributes> [Relationships]

In order to derive a pattern language discovered pattern instances represent pattern language vocabulary. The structural composition of pattern elements and their relationships (governing pattern structure and relations - meta-model cf. Figure 7.2) represents a language grammar as expressed in Listing 7.1 - pattern template. A formal template-based specification of discovered patterns allow us to specify the relations that exist among patterns in the language.

For an additional discussion of graph vs UML based modelling of the pattern specification please refer to Chapter 2. We prefer a graph-based template for pattern specification:

- *Establishing Static and Dynamic Relationships -* in contrast to some predefined relationships among patterns [Porter 2005, Zdun 2007], a graph-based modelling [Ehrig 2004] allows capturing the semantics of pattern relationships. More specifically, an attributed graph is represented as pattern language with individual patterns as attributed nodes and pattern relationships as attributed edges of a graph (Listing 7.1). An individual pattern is represented as a graph node while the directed edge represents a static or a dynamic relation among the adjacent nodes (connected patterns).

- *Pattern Matching and Selection -* if individual patterns are represented as graph nodes, we can exploit sub-graph isomorphism [Jiang 2012] (based on node matching) to select individual patterns (i.e., nodes) from templates (i.e., graphs).

- *Visualising Pattern Composition and Relations -* enables abstracting a complex pattern hierarchy. Pattern visualisation greatly facilitates analysing pattern structures to evaluate possible consequences and alternatives in a given evolution context.

- *Graph Network of Patterns -* define possible relationships among patterns in the language. Graph-based structure provides a flexible mechanism to search and retrieve patterns efficiently.

7.6.1 Mapping Elements of Template for Graph-based Pattern Specification

A template-based specification of architecture change patterns as a typed attributed graph is expressed as 6-tuple: Template $=< G_{TMP}, N_{CLS}, G_{PAT}, N_{CMP}, E_{SEQ}, N_{REL} >$ summarised in Table 7.4.

The meta-model for pattern-based architecture evolution only represents a structural composition of change patterns. In order to enable compositional semantics for pattern elements, we also explain the binary composition relationships of pattern elements. For example, the relation $Pattern \xleftarrow{isComposedOf} Operations$ represents that a pattern is composed of architecture change operations.

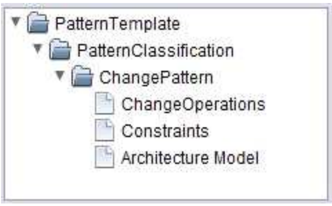
Graph Element	Pattern Element	Description
Outer Graph	G_{TMP} - Pattern Template	Pattern template (PAT_{TMP}) is represented as an outer graph that contains pattern classification, composition and variants as a nested graph structure.
Node	N_{CLS} - Pattern Classification	Pattern classification (CLS) is represented as an outer node of the graph to contain patterns and their possible variants.
Nested Graph	G_{PAT} - Pattern Structure	Hierarchical composition of pattern (PAT) is represented as a nested graph.
Nested Node	N_{CMP} - Pattern Composition	Pattern composition elements including the possible variants (PAT_{var}) are represented as a set of nested nodes in pattern graph.
Edge	E_{SEQ} - Pattern Sequence	The Outer edge represents a possible interconnection among the different classifications in the template to create a sequence of patterns (PAT_{seq})
Nested Edge	N_{REL} - Pattern Relations	Nested edge represents the binary relationships among change patterns elements (e.g: $PAT < Evolves > ARCH$)
 PatternTemplate PatternClassification ChangePattern ChangeOperations Constraints Architecture Model		

Table 7.4: Graph-based Representation of Change Patterns Template.

1. **[Classifies :** $CLS \xleftarrow{Classifies} PAT$**]** - defines the classification of change pattern instances in the pattern template. Pattern classification therefore defines a logical grouping of change patterns based on their impact of change on architecture model (addition, removal, modification etc.). It is based on pre-defined categories including $=< Inclusion, Exclusion, Replacement >$ change with classification $id = 1, 2, 3$ respectively. Currently this classification is based on a manual analysis of identified patterns corresponding to their impact on architectural elements (Line 4 - 34, Listing 7.1).

2. **[ComposedOf :** $PAT \xleftarrow{ComposedOf} OPR$**]** - defines the change operational composition in a given pattern instance. Note that a set of change operators perform the architectural changes when a pattern is applied to a given architecture model (Line 21 - 24, Listing 7.1).

3. **[ConstrainedBy :** $PAT \xleftarrow{ConstrainedBy} CNS$**]** - define a set of constraints on change patterns. The constraints ensure the structural integrity of architecture model before and after change pattern application (Line 16 - 20, Listing 7.1).

4. **[Evolves :** $PAT \xleftarrow{Evolves} PAT$**]** - defines the application of a change pattern on a given architecture model. Please note that, CBSAs defines application domain of the discovered change patterns. Therefore, the applicability of patterns and consistency of pattern-based evolution beyond architectural descriptions for CBSAs is not guaranteed (Line 11 - 15, Listing 7.1).

5. **[hasVariant :** $PAT \xleftarrow{hasVariant} VAR$**]** - defines the relationship among a pattern and its possible variants. The variant of a pattern has the same structure and semantics of a pattern, however a variant represents the variations among the possible implementations of a pattern. It only requires the identification ($PatternID$) of possible variants of a given change pattern (Line 26 - 31, Listing 7.1).

6. **[follows :** $PAT_i \xleftarrow{follows} PAT_j$**]** - defines the sequence of two change patterns PAT_i follows PAT_j. In order to develop a pattern system, patterns in the language has to be applied in a specific order defined by one or more pattern sequences. Depending on the context in which the pattern language is applied, there can be several sequences in a pattern language. Specifically, the sequence $< PAT_i \prec PAT_j \prec PAT_k >$ means: pattern PAT_i is selected before pattern PAT_j, which itself is selected before PAT_k (Line 35 - 36, Listing 7.1).

7.6.2 Semi-automated Specification of Change Patterns in the Template

The discovered patterns are specified in the pattern template in a semi-automated fashion. By semi-automated we mean the prototype-based tool support for pattern specification along with the necessary user intervention to guide the pattern specification process. The details of the prototype for change pattern specification along with user intervention are provided in **Appendix D**. More specifically, the *change operations*; *constraints* along with the *impact of change pattern* on the architecture model is provided by the prototype. User can view such provided information to specify an appropriate *name* the *intent* of each change pattern.

Listing 7.1: Graph-based Template for Pattern Specification (GraphML notation [Brandes 2002a]).

```
1   <graphml>
2     <graph id="PatternTemplate" edgedefault="directed">
3       <desc> Graph-based Representation of Change Pattern Template </desc>
4         <node id = "CLS1">
5           <desc> The graph node provides pattern classification </desc>
6             <graph id="ChangePattern" edgedefault="directed">
7               <desc> A nested graph to represent individual pattern </desc>
8                 <node id = "PAT1">
9                   <desc> Pattern specific details </desc>
10                </node>
```

```
              <node id = "ARCH1">
                <desc> Represent the architecture elements affected </desc>
              </node>
              <edge id = "Evolves" source = "PAT1" target="ARCH1">
                <desc> Pattern Evolves Architecture </desc>
              <node id = "CNS1">
                    <desc> Specifies the enforced constraints </desc>
              </node>
              <edge id = "isConstrainedBy" source = "PAT1" target="CNS1">
                <desc> Pattern Constrained By Constraints </desc>
              <node id = "OPR1">
                <desc> Representing the Change Operationalisation </desc>
              </node>
              <edge id = "isComposedOf" source = "PAT1" target="OPR1">
                <desc> Pattern Composed of Operators </desc>
              <node id = "VAR1">
                <desc> Possible pattern variants </desc>
                ...
              </node>
              <edge id = "hasVariant" source = "PAT1" target="VAR1">
                <desc> Pattern has a Variant </desc>

         </graph>
       </node>
         <edge id = "Follows" source = "PAT1" target="PATN">
         <desc> Pattern PAT1 follows another pattern PATN </desc>

   </graph>
</graphml>
```

7.7 Chapter Summary

In this chapter, based on the change log graph (Chapter 5) and change operationalisation (Chapter 6) we discovered architecture change patterns. In this chapter, we provide an answer to **RQ 2** that requires the development of methods and techniques for discovering and specifying architecture evolution-reuse knowledge (operators and patterns) from architecture evolution histories.

We present the architecture change patterns discovery from logs as a modular (each algorithm representing a module that can be integrated) solution with appropriate parametrisation, user intervention and (semi-) automation for discovery algorithms. The discovered pattern instances and their variants (in Section 4) represent the *vocabulary of pattern language* that evolves over time. We proposed pattern discovery as a continuous process by mining new change patterns from different logs. Furthermore, based on a meta-model for pattern-based evolution, we derived the *pattern language grammar* that provides the structural composition and relations among pattern elements. The primary contribution of this chapter are algorithms for change pattern discovery from change log graphs. This chapter serves as a pre-requisite to Chapter 8, that aims at establishing possible relations - *pattern language sequencing* - among discovered patterns to derive a pattern language.

Chapter 8

Composition and Application of a Change Pattern Language for Architecture Evolution

Contents

8.1 Chapter Overview

We exploited change logs with graph-based modelling and discovery of architecture change patterns that represent generic and reusable solutions to recurring evolution problems in Chapter 7. However, the true potential for individual change patterns can only be achieved; if patterns are applied in the context of each other - by establishing pattern interconnections - known as a system or language of patterns [Goedicke 2002, Alexander 1999, Buschmann 2007]. In the software engineering domain, there is a growing need to develop a pattern community[1,2] to collaborate on discovering innovative patterns and deriving pattern languages. The primary aim of a community-oriented effort is to promote patterns as reusable artefacts to address different phases of a software life-cycle including architectural design and evolution [Lytra 2012, Goedicke 2002].

The primary contribution of this chapter is to explain the *composition* and illustrate the *application* of a change pattern language that supports pattern-driven reuse in architectural evolution. It is vital to mention that: *unlike a programming language that provides an executable syntax, a pattern language provides a generic vocabulary; a grammar as well as a sequence of applications of a collection of individual patterns* [Zdun 2007]. Therefore, the concept of a pattern language is fundamentally inspired by the composition of a natural language [Porter 2005, Zdun 2007] that has a vocabulary, grammar and sequences of words to enable communication. In the proposed architecture change patterns language, the *vocabulary* is represented as a collection of discovered patterns and their possible variants (cf. Chapter 7). This means the vocabulary of the proposed pattern language evolves over time with the discovery of new change patterns. The language *grammar* governs the rules and structure of relations of individual patterns in the language. Finally, the pattern *sequencing* determines an ordered application of architecture change patterns in the language. The order of the pattern application determines which patterns have to be applied before or after the application of other patterns. In this chapter we aim to answer **RQ 4** (cf. Chapter 1) that aims to investigate patterns for reuse in architecture evolution.

Our solution to derive a pattern language with change patterns is also inspired by Alexander's seminal theory [Alexander 1999, Alexander 1979] about pattern languages that integrate patterns as repeatable solution to build complex architectures in real world. A language-based pattern collection facilitates an iterative pattern selection and their application to enable an incremental evolution in architectures. By incremental evolution we mean: *decomposing evolution process into a manageable set of evolution scenarios that could be addressed in a step-wise manner* - assuming each

[1] Pattern Languages of Programs www.hillside.net/plop/2013/

[2] European Conference on Pattern Languages of Programs www.europlop.net/

pattern provides a (reusable) solution to a given evolution scenario [Goedicke 2002]. As detailed in Chapter 3, evolution-reuse knowledge [Zhang 2012] in the proposed pattern language is expressed as a formalised collection of interconnected patterns.

8.2 Overview of Pattern Language Composition and Application

In order to support pattern interconnections (a.k.a. *system-of-patterns* [Buschmann 2007]) and to derive a pattern language for architecture evolution, an overview of the proposed solution in presented in Figure 8.1. We present a layered solution to distinguish between the aspects of composition and application of a change pattern language. In Figure 8.1, we map our proposal of a pattern language to the generic processes: architecture change mining and architecture change execution in the PatEvol framework (cf. Chapter 4) and discussed below. We also introduce a structural representation for pattern language composition.

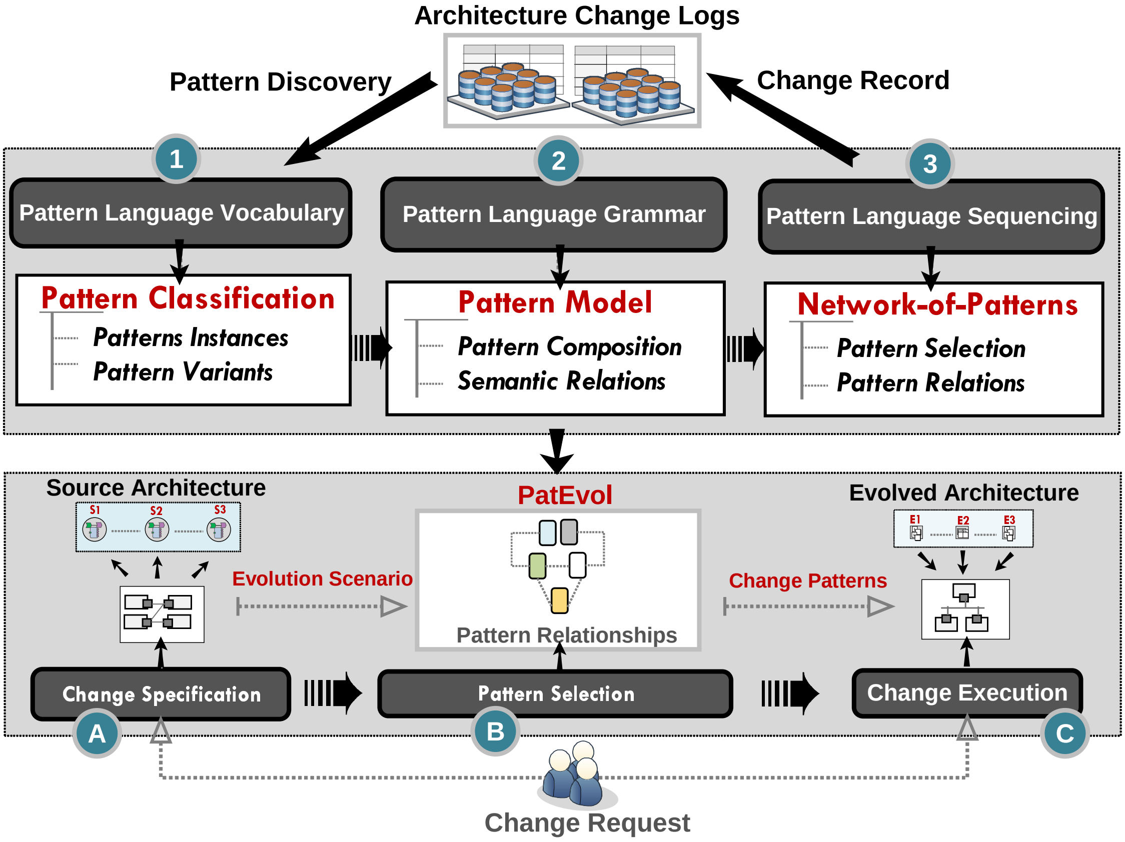


Figure 8.1: A Layered Overview of the Proposed Solution.

In the existing literature details about the structure of pattern language for architectural development and maintenance are presented in [Zdun 2007, Goedicke 2002, Lytra 2012]. We follow the guidelines for language composition from [Zdun 2007] to derive a change pattern language comprising of: a) a classified composition of patterns and their variants (1. $Vocabulary : V_{PatEvol}$) along with a b) set of rules that govern the structure and relations among pattern elements (2. $Grammar : G_{PatEvol}$) to create a c) sequence-of-patterns (3. $Sequencing : N_{PatEvol}$). In the following, the notation $\triangleright$ denotes a containment relation (e.g. a classification contains patterns and their possible variants) and $\prec$ denotes an order relation (e.g. a pattern PAT_1 follows another pattern PAT_2). We formalise the structural composition of the pattern language as: $PatEvol(V_{PatEvol} \times G_{PatEvol} \times S_{PatEvol})$ =

$$\left[\begin{array}{c}
V_{PatEvol} = \{CLS \triangleright PAT(_{var_1,\ var_2,\ \dots\ var_3})\} \dots (1) \\
\\
\begin{aligned}
G_{PatEvol} = \{&PAT_{<ClassifiedBy>}CLS, \\
&PAT_{<ComposedOf>}OPR, \\
&PAT_{<ConstrainedBy>}CNS, \\
&PAT_{<Evolves>}ARCH \\
&PAT_{<hasVariant>}VAR\} \dots (2)
\end{aligned} \\
\\
S_{PatEvol} = \{PAT_1 < var_1, \dots, var_n > \quad \prec \dots \prec PAT_n < var_1, \dots, var_n >)\} \dots (3)
\end{array}\right]$$

We now map our proposal of a pattern language in Figure 8.1 to the generic processes of architecture change mining and architecture change execution in the PatEvol framework. .

8.2.1 Architecture Change Mining for Pattern Language Composition

In Figure 8.1 we exploit architecture change mining to derive reuse knowledge in terms of *change operationalisation*, *operational dependencies* and *discovered pattern* instances from logs.

1. **Vocabulary of the Pattern Language -** we investigate architecture change logs to discover a classified composition of *change patterns* and possible *variants* (i.e., vocabulary) in Figure 8.1. We have already provided the details about discovered change patterns and their variants in Chapter 7. A pattern language vocabulary continuously evolves with discovery of new change patterns over time whenever pattern discovery algorithms are executed on change logs.

2. **Grammar of the Pattern Language -** it express the structural composition of pattern elements that also governs the relationships among pattern elements (i.e., grammar). The grammar is derived based on the *change patterns meta-model* and *binary composition relationships* between pattern model elements from Chapter 7.

3. **Pattern Sequencing in the Language -** in a language-based collection, the benefits associated to a set of related patterns are more than the sum of the benefits of each individual pattern [Goedicke 2002, Porter 2005]. More specifically, in a language context we establish relationships or an order of application for individual patterns to be applied in a sequential fashion from a collection [Zdun 2007, Porter 2005]. For illustrative purposes we exemplify a pattern relation in Figure 8.2 - more concrete examples are provided later in the chapter. In Figure 8.2 we represent a generic relation (REL_1) among two pattern PAT_1 and PAT_2. The relation REL_1 specifies that PAT_1 must be applied before PAT_2 during architecture evolution. If there is a scenario such that a) first a component needs to be integrated in the architecture as a mediator and b) later it is decomposed into further fine-grained components; we can specify this pattern relation as ComponentMediation *follows* FunctionalSlicing as in Figure 8.2.

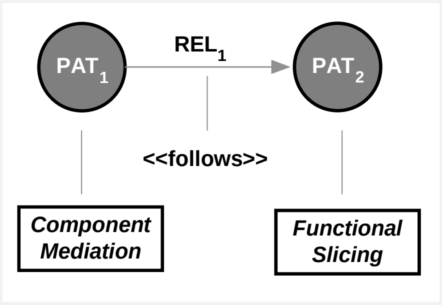


Figure 8.2: An Overview of the Relation between Change Patterns.

Pattern interconnection requires the creation of either static, dynamic or both types of relations between change patterns. *Static* or predefined relations express specialised and generalised type patterns in the language. However, due to an unanticipated nature of architecture change; static relations are limited when expressing sequential relations between the patterns in the language. In contrast, sequential or *dynamic*[3] relation determines if a pat-

[3] Please note that our views about dynamic sequences are consistent with the pattern community's view on pattern and pattern languages [Zdun 2007, Porter 2005]: patterns and pattern languages are living documents/artefacts that evolve over time as new pattern, their variants or dependencies to other patterns emerge. Our solution with architecture change mining process supports a continuous discovery of new patterns (and ultimately new relations) from logs over time.

tern is dependent-on or independent-of other patterns in the language. In an ideal context, patterns language must support dynamism in creation or destruction of pattern relations that is driven by the context of evolution. Creating a sequential relationship among change patterns in the language is discussed in **Section 8.3**.

8.2.2 Architecture Change Execution for Pattern Language Application

It refers to exploiting the patterns and their relations in a pattern language to address the evolution scenarios by mapping the problem view with solution view of the domain (i.e., CBSAs and their evolution). In Figure 8.1, we propose architecture change execution (application of pattern language) to enable pattern-driven reusable evolution of component-architectures.

1. **Pattern Selection from Language Collection -** is a significant challenge for inexperienced developers or architects due to a) searching of required patterns in an ever growing collection and, b) selecting the appropriate patterns or possible alternatives [Kampffmeyer 2007]. In this and similar situations of pattern selection, systematically applying patterns requires a certain amount of expertise from the software architect or designer. More specifically, the architect/designer has to well understand how a pattern's solution fits into the overall architecture and how other patterns can be applied to resolve new or open issues as a consequence of applying the first pattern [Zdun 2007]. Change patterns require certain expertise from the architects in terms of mapping the problem-solution for the domain in which the patterns should be applied. Some typical questions that arise could be:

    - Which pattern should I choose first?
    - Which variant of the pattern works best?
    - Which pattern should be applied next?

2. **Pattern Application in Architecture Evolution** To support pattern-based architectural evolution, we propose to specify the architectural changes (as addition, removal, or modification) of elements in existing CBSA. Our solution follows the idea of a declarative specification of the changes (guided by [Sadou 2005]) that enables the selection of appropriate pattern sequences to derive reusable evolution strategy based on given evolution scenarios. Also, a pattern language provides a method of systematic reuse based on an incremental application of patterns from a collection [Goedicke 2002].

8.3 Pattern Relations as the basis for Language Composition

Once the grammar (pattern composition and binary relations) and vocabulary (patterns and their variants) are specified, we can establish pattern relationships (a.k.a. sequencing of patterns) presented in Figure 8.4. More specifically, a pattern language provides a topology to derive sequences, similar to the natural language where the grammar provides the structure for generating sentences. An important question arises: *'why we choose a particular sequence of patterns from the possible alternatives in the language*?'. In the literature, pattern sequencing is derived based on a pattern hierarchy [Porter 2005] (e.g., large patterns must be on top of smaller patterns) or using the annotated grammar [Zdun 2007] with the Question Option Criteria methodology [MacLean 1991].

8.3.1 Establishing the Pattern Relations

A pattern language provides a topology to derive sequences, similar to the natural language where the grammar provides the structure for generating sentences. In the context of defining pattern relations, we must determine: 'why we choose or define a particular sequence of patterns (relations among patterns) from the possible alternatives in the pattern language (available collection)?' In the literature, pattern relations are derived based on the analysis or observations of the application domain of the patterns. For example, in the design and development of a composable software systems (i.e., application domain) the pattern relations are defined based on a pattern hierarchy such that the large patterns must be on top of smaller patterns [Porter 2005]. Also, in [Zdun 2007] architectural design scenarios and their potential solution are analysed to derive relations among the patterns using Question Option Criteria methodology [MacLean 1991].

In our solution, we define the pattern relations as illustrated in Figure 8.4 based on the application domain of the proposed pattern language - evolution of the component-based architectures. Specifically, we have analysed the architectural evolution based on individual architectural changes and change patterns from logs to define pattern relations. We define the relations manually during change pattern specification in the template. For example, during pattern specification we define the relation *hasVariant* between *ComponentMediation* and *ParallelComponentMediation* as the later pattern is a variant of the former one. It is vital to mention that in existing [Porter 2005, Goedicke 2002] and our solution the definition of pattern relation is a manual and subjective process [Zdun 2007], i.e., the pattern author decides âĂŞ based on a number of factors (such as based on personal experience, domain analysis) âĂŞ about selection of a specific relation among patterns.

8.3.2 Static Sequence of Patterns

In a pattern language context, a static sequence of patterns represents a pre-determined or a fixed relation among two or more patterns [Porter 2005]. A static sequence is determined with a manual analysis of the domain (analysis of architecture evolution scenarios) to create a fixed relation among the patterns. An example of the static sequencing is provided in Figure 8.4. Details about individual patterns are already provided in Chapter 7 and **Appendix D**. In Figure 8.4 we derive a sequence that is interpreted as ComponentMediation $< follows >$ ActiveDisplacement: "if the replacement of a component is required". A pattern sequence annotation in the language (Listing 8.1) is given as:

Listing 8.1: Static Pattern Sequence (ComponentMediation $< follows >$ ActiveDisplacement).

```
<graphml>
  <graph id="ChangePattern" edgedefault="directed">
    <node id = "ComponentMediation">
      -- Pattern Elements Here --
    </node>
    <edge id = "Follows" source = "ComponentMediation" target="ActiveDisplacement">
    <node id = "ActiveDisplacement">
      -- Pattern Elements Here --
    </node>
  </graph>
</graphml>
```

Limitations of Static Sequences - A static sequencing is a rigid structure of the pattern language with a minimal flexibility. This could be particularly limiting in a context where the exact sequences of patterns depend on some arbitrary evolution scenario. For example, in many situations we might need the application of ActiveDisplacement pattern directly preceding ComponentMediation. In addition, as we discover new patterns and integrate them with existing pattern collection in the language new relationships among patterns emerge or the older ones evolve. This and many other similar situations ignore the static sequence, in fact a static-sequence only represents a specific organisation of patterns that must be dynamically adjusted.

8.3.3 Dynamic Sequence of Patterns

In contrast to static sequencing, a dynamic sequence of patterns represents the dynamic relationships among patterns that evolve based on the context of pattern application [Porter 2005]. This means, during change execution (runtime) there does not exist any relation among patterns, instead patterns are dynamically selected one after another by means of an iterative specification of the given evolution scenarios. To create dynamic sequencing, we follow a step-wise process by exploiting design-space analysis for patterns [Zdun 2007, MacLean 1991] that is based on the Question Option Criteria [MacLean 1995]. For example, in Figure 8.3 the intent of a pattern language user is to enable component composition, but he/she is unclear which pattern to select. By following the QOC method, patterns can be applied in an arbitrary sequence (selecting one pattern at a time). Moreover, the QOC process is iterative such that after selection of a specific pattern. A new question (i.e., evolution scenario) can be specified to select the next patterns. For example, in contrast to a static sequence the selection of the Component Mediation pattern based on QOC methodology is achieved as follows.

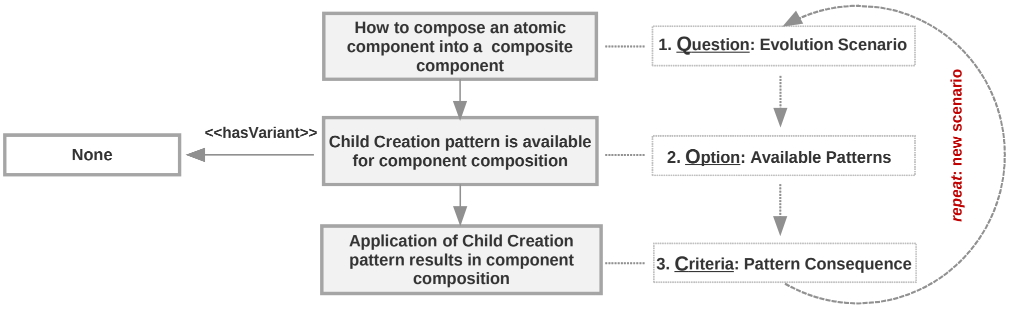

How to compose an atomic component into a composite component
Child Creation pattern is available for component composition
Application of Child Creation pattern results in component composition
<<hasVariant>>
None
1. Question: Evolution Scenario
2. Option: Available Patterns
3. Criteria: Pattern Consequence
repeat: new scenario

Figure 8.3: Overview of the QOC Methodology for Dynamic Selection of Patterns.

1. **Question** - How to compose an atomic component into a composite one?

2. **Option** - ChildCreation pattern enables component composition.

3. **Criteria** - The consequence of ChildCreation pattern is the composition of an atomic component into a composite component (composed of one or more child/sub component(s)).

We generalise the dynamic sequencing of patterns in the language as follows, where $\prec$ represents a sequencing operation, *PAT* represents a selected pattern instance and *var* represent the possible variant(s) of a pattern:

$$QOC := \{PAT_1 < var_1, \ldots, var_n > \prec \ldots \prec PAT_n < var_1, \ldots, var_n >)\}$$

We further clarify and exemplify QOC-based pattern selection in **Section 8.6**.

8.3.4 Pattern Variants

The composition of a given pattern instance may vary depending on possible variations in architectural changes. This means, there exist variations in the composition of a pattern and that is expressed as pattern variants. Variants allow to model and evaluate the possible variation that may exist in the problem-solution space for pattern-based architecture evolution. For example (in Chapter 7), while discovering change patterns we observed exact and inexact matching among discovered pattern instances - representing a variation of change patterns. More specifically, we presented the Component Mediation pattern having two possible variants Parallel Mediation and Corelated Mediation. In a pattern language context, relation among a pattern PAT_j and its variant VAR_k generalised as $VAR_k \xleftarrow{hasVariant} PAT_j$ and is expressed:

Listing 8.2: Pattern Variants of ComponentMediation Pattern.

```
<graphml>
  <graph id="ChangePattern" edgedefault="directed">
    <node id = "ComponentMediation">
      — Pattern Elements Here —
    </node>
    <edge id = "hasVariant" source = "ComponentMediation" target="ParallelMediation">
    <node id = "ParallelMediation">
      — Pattern Elements Here —
    </node>
        <edge id = "hasVariant" source = "ComponentMediation" target="CorelatedMediation">
    <node id = "CorelatedMediation">
      — Pattern Elements Here —
    </node>
  </graph>
</graphml>
```

In the work on design patterns [Gamma 2001] (with more than a decade of research and practice), pattern variants have emerged and become almost a complementary solution to the original GOF patterns. In contrast, the change patterns discovered in this thesis are comparatively much less mature. Therefore, we cannot claim (without further validations) that every change pattern has (an empirically discovered) variant as an integral part of the pattern.

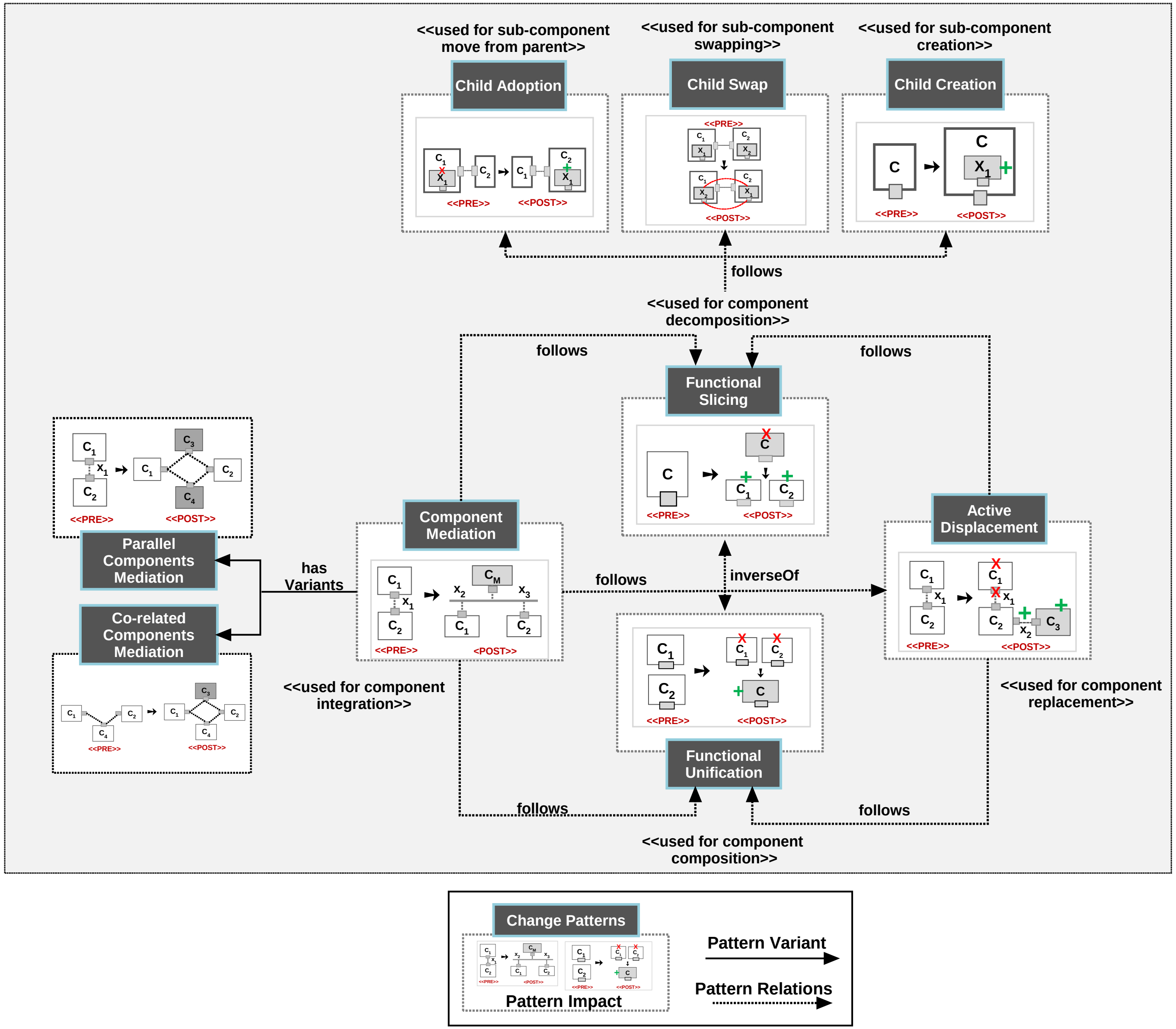

<<used for sub-component move from parent>>
Child Adoption
<<PRE>>
<<POST>>
<<used for sub-component swapping>>
Child Swap
<<used for sub-component creation>>
Child Creation
follows
<<used for component decomposition>>
follows
follows
Functional Slicing
Component Mediation
Active Displacement
Parallel Components Mediation
Co-related Components Mediation
has Variants
follows
inverseOf
<<used for component integration>>
<<used for component replacement>>
Functional Unification
follows
follows
<<used for component composition>>
Change Patterns
Pattern Impact
Pattern Variant
Pattern Relations

Figure 8.4: An Overview of Change Pattern Language.

8.4 Application Domain of Change Pattern Language

The pattern language embodies its knowledge by investigating change representation in evolving architecture models. Therefore, the applicability (a.k.a. application domain) of the proposed pattern language is limited to component-based architecture models (CBSA) [Medvidovic 1999, van der Aalst 2002] and evolution in CBSAs [Garlan 2009, Le Goaer 2008]. We have already detailed graph-based modelling[4] of component-based software architectures in Chapter 2. In this section we clarify the application domain of proposed pattern language. Therefore, we provide a quick review of graph-based descriptions of CBSAs and also discuss some architecture evolution scenarios that are used to illustrate pattern-driven evolution. Additional details about the component-based architecture model are provided in **Appendix D**.

8.4.1 Evolution in Component-based Software Architecture

We look at two evolution scenarios to demonstrate the desired changes in existing architecture model for the EBPP case study [EBPPCaseStudy]. We adopt the Architecture Level Modifiability Analysis (ALMA) [Bengtsson 2004] method for scenario elicitation and analysis of EBPP architecture evolution. Further details about the EBPP case study are presented in **Appendix B**. Based on the ALMA methodology, we follow a three step process for selection, evaluation and interpretation of evolution scenarios.

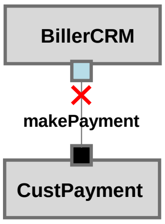
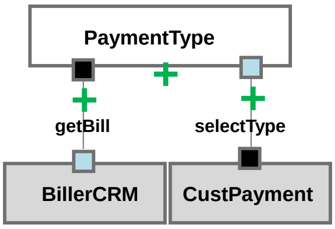
Two evolution scenarios are presented in Table 8.1. Key characteristics and evolution-centric aspects of the component-driven architecture are:

- *Composite Change Execution* that must abstract atomic change operations $[Add, RemoveModify]$ on architecture elements $< Component, Connectors >$ into composite changes that allow $[\ldots, Integration, Composition, Replacement, \ldots]$ of a set of architecture elements.
- *Evolution Reuse* is a key characteristic that must enable a generic, reuse-driven change for recurring evolution problems [Garlan 2009, Le Goaer 2008] in component-driven architecture models.
- *Consistency of Evolving Architecture Models* ensures that the structural integrity and composition constraints of an architecture model are preserved before (*preconditions*) and after evolution (*postconditions*).

[4] Additional details about graph modelling for architecture description [Baresi 2002] is provided in a report `www.computing.dcu.ie\~aaakash\GraphModel.pdf`

Evolution Scenario I	
A. Scenario Selection	**B. Scenario Evaluation**

ES1 - [. . .] to integrate a mediator component PaymentType that facilitates the selection of a payment type mechanism among the directly connected components BillerCRM and CustPayment.	Architecture is modified with addition of a new components and two connectors to mediate customer billing and payments:
	opr1:= Add($PaymentType \in CMP$)
	opr2:= Add($getBill, selectType \in CON$)

C. Results Interpretation	
Change Preconditions	Change Postconditions
	 PaymentType getBill selectType BillerCRM CustPayment
Evolution Scenario II	
A. Scenario Selection	**B. Scenario Evaluation**

ES2 - [. . .] To compose the PaymentType component with DirectDebit and CreditPayment child components that a customer to avail-of flexible options for billing payments.	The internal architecture of PaymentType is modified with addition of two child components DirectDebit and CreditPayment
	opr1:= Add($DirectDebit \in CMP$)
	opr2:= Add($CreditPayment \in CMP$)

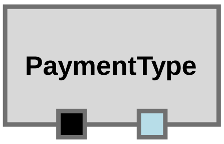
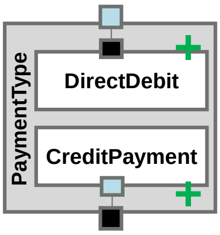
C. Results Interpretation	
Change Preconditions	Change Postconditions
	

Table 8.1: Selection, Evaluation and Interpretation of Architecture Evolution Scenarios.

8.5 Graph Transformation for Architecture Evolution

Graph transformation provides a mathematical foundation to evolve architecture models that are represented as graphs [Carrière 1999, Baresi 2006b]. Therefore, in this section we focus on presenting the foundational concepts of graph transformation before presenting details about architecture evolution guided by change patterns. The technical details about (graph-) transformation-driven evolution are presented in [Baresi 2002] with a step-by-step approach to enable graph transformation for software evolution. More specifically, in the context of architecture evolution the work in [Carrière 1999] exploits graph transformation for architectural re-engineering and evolution. In addition considering attributed graph, [Baresi 2006b] focused on attributed graph transformation for evolution and refinement of service-driven architecture that is similar to our solution.

In contrast to some of the well-known graph transformation solutions for architecture evolution [Carrière 1999, Baresi 2006b], we focus on architectural transformation that is guided by architecture change patterns. In this context, the architecture model is represented as an attributed graph [Ehrig 2004], and we exploit an attributed graph transformation for architecture evolution [Baresi 2006b]. In order to enable architecture evolution, we follow the double push-out (DPO) as an algebraic approach for graph-transformation [Loewe 1997]. Technical details about algebraic graph transformation and mathematical foundations for DPO graph transformation are discussed in [Ehrig 2006, Loewe 1997] with tool support provided in [Taentzer 2000].

The DPO approach enables specification of a graph transformation system that enables changes in a graph architecture in the form of pairs of graph morphisms $\left\langle Source \xleftarrow{src} Intermediate \xrightarrow{trg} Target \right\rangle$ as illustrated in Figure 8.5. We utilise a hypothetical example in Figure 8.5 and introduce a DPO-based approach to architecture evolution that is used to achieve pattern-based architecture evolution later in Section 8.6. In the example Figure 8.5, we want to add a new graph node (N_x) between connected nodes (N_1, N_2) such that N_x is an intermediate node between N_1 and N_2

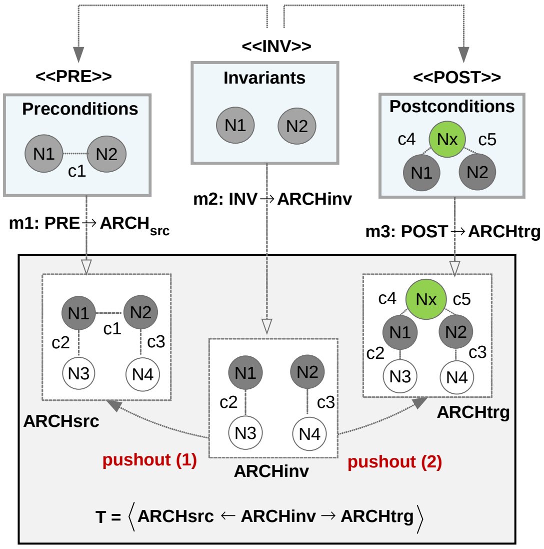


Figure 8.5: Overview of Double Push Out Graph Transformation System.

In Figure 8.5, we express a graph transformation system for architecture evolution as follows: $G_{trans} =< ARCH, CNS, OPR >$ consists of an architecture model (ARCH) represented as a typed graph to define the architectural elements and their relationships, a set of constraints (CNS) to further restrict the valid architecture models, and a set change operations (OPR) as graph transformation rules.

8.5.1 Graph-based Architecture Models

In a graph transformation system architecture models (ARCH) consists of:

$$\langle ARCH_{src}, ARCH_{inv}, ARCH_{trg} \rangle \in ARCH$$

1. *Source Architecture Model* is represented as $ARCH_{src}$, that is evolved towards

2. *Target Architecture Model* represented as $ARCH_{trg}$, using an

3. *Invariant Architecture Model* represented as $ARCH_{inv}$ that is preserved during change, $ARCH_{inv}$ is an intermediate architecture (graph) $ARCH_{src} \xleftarrow{src} ARCH_{inv} \xrightarrow{trg} ARCH_{trg}$ between source and target.

The architecture graph is specified as five tuple $ARCH = (N_G, E_G, N_A, E_{NA}, E_{EA})$ (cf. Chapter 2). In this context, N_G and N_A are called the graph and attribute nodes (for representing architectural components and their attributes), respectively while N_A, E_{NA}, and E_{EA} are called the graph, node attribute, and edge attribute edges, respectively. E_G represents graph edges as connectors in architecture. For example, in Figure 8.5, $ARCH_{src}$ represents the configuration of source architecture having nodes (components) N_1, N_2, N_3, N_4 are connected using the edges (connectors) c_1, c_2, c_3. The source graph $ARCH_{src}$, invariant graph $ARCH_{inv}$ and the target graph $ARCH_{trg}$ represent an instance of ARCH, therefore $ARCH_{src}, ARCH_{inv}, ARCH_{trg} \in ARCH$.

8.5.2 Constraints on Graph Model

The constraints (CNS) on a graph model refer to a set of pre-conditions (PRE) and post-conditions (POST) to ensure the consistency of graph models during transformation. In addition, the invariants (INV) ensure structural integrity of individual architecture elements during change execution. We specify transformation constraints as:

1. *Transformation Preconditions* represent the context of graph elements before transformation. For example in Figure 8.5, PRE specifies the exact sub-graph $c_1(N_1, N_2)$ that is subject to change in the original graph $ARCH_{src}$. In order to apply the transformation rule, we must find a match $m1$ of the PRE in $ARCH_{src}$, such that $m_1 : PRE \rightarrow ARCH_{src}$ provides a structural matching between PRE and $ARCH_{src}$ based on node and edge labels in Figure 8.5.

2. *Transformation Postconditions* specify the context of a transformed graph as a result of the change execution. After applying changes on specified elements the overall graph structure must be preserved. For example, $POST$ specifies the exact sub-graph $c_4(C_1, C_X)$ and $c_5(C_2, C_X)$ that is added to the original architecture $ARCH_{src}$, in Figure 8.5. In order to

include the modified architecture elements $POST$ in the target graph $ARCH_{trg}$ an exact structural match m_3 of $POST$ in $ARCH_{trg}$ must exist such that $m_3 : POST \rightarrow ARCH_{trg}$.

3. *Transformation Invariants* expressed as INV and $ARCH_{inv}$ (or the invariant graphs) that provide the common interface for $ARCH_{src}$ and $ARCH_{trg}$ which is preserved during transformation, in Figure 8.5. It describes a part of the graph (intersection of $ARCH_{src}$ and $ARCH_{trg}$) part which has to exist to apply the transformation rule. This is represented as Figure 8.5, the invariant graph $ARCH_{inv}$ as $m_2 : INV \rightarrow ARCH_{src}$ with Double-Push-Out (DPO) graph transformations [Loewe 1997].

8.5.3 Graph Transformation Rule

The transformation rules allow a declarative specification of graph transformations (by adding or removing the specified nodes and edges) on a source graph $ARCH_{src}$ resulting in a target graph $ARCH_{trg}$. A transformation rule T is given as: $T = \left\langle ARCH_{src} \xleftarrow{src} ARCH_{inv} \xrightarrow{trg} ARCH_{trg} \right\rangle$. For example in Figure 8.5, the application of the graph transformation requires finding a match PRE in the source graph $ARCH_{src}$ with $c1(N_1, N_2)$ as the common sub-graph in PRE and $ARCH_{src}$.

1. *Pushout I - Deletion of an edge* - PRE INV represents the architecture element(s) which is to be deleted from $ARCH_{src}$. For example, in Figure 8.5 the edge c_1 is removed from $ARCH_{src}$ among nodes N_1 and N_2. The invariant graph $ARCH_{inv}$ is obtained from the source graph $ARCH_{src}$ by removing $c1$ from $ARCH_{src}$.

2. *Pushout II - Addition of nodes and edges* - POST INV represents the sub-graph which needs to be added in $ARCH_{src}$ to obtain $ARCH_{trg}$ during change execution. For example in Figure 8.5, the node N_x is added with edges c_4 and c_5 in $ARCH_{inv}$ to obtain $ARCH_{trg}$.

Graph Pattern - a formal graph-based notation for change pattern (PAT) is provided in Chapter 7 with its concrete representation as graph modelling notation [Brandes 2002a]. We express graph pattern as $PAT < name, intent >$: $PRE(ARCH_{src}) \xrightarrow{ARCH_{inv}} POST(ARCH_{trg})$ as a constrained composition of source to target architectural transfromations that can be reused. The application of graph patterns for transformation requires finding a match PRE in the source graph $ARCH_{src}$ and replacing preconditions with post-conditions $POST$ that leads the target graph $ARCH_{trg}$ by application of graph transformation and preserving the invariant structure $ARCH_{inv}$.

8.6 Application of Change Pattern Language

After presenting pattern relations in the language and application domain of pattern language, we now focus on pattern-language support for evolution in component-based architectures. An overview of the solution for pattern-based architecture evolution is provided in Figure 8.6 - a detailed view of the architecture change execution process (from Section 8.2). In the remainder of this section, first we discuss the design space analysis for change pattern selection in **Section 8.6.1**. We provide a three-step process including *change specification*, *pattern selection* and *change execution* to enable architecture evolution in **Section 8.6.2**. Change patterns from the language could be selected and applied in a sequential fashion to support an incremental evolution [Garlan 2009, Le Goaer 2008] in CBSAs.

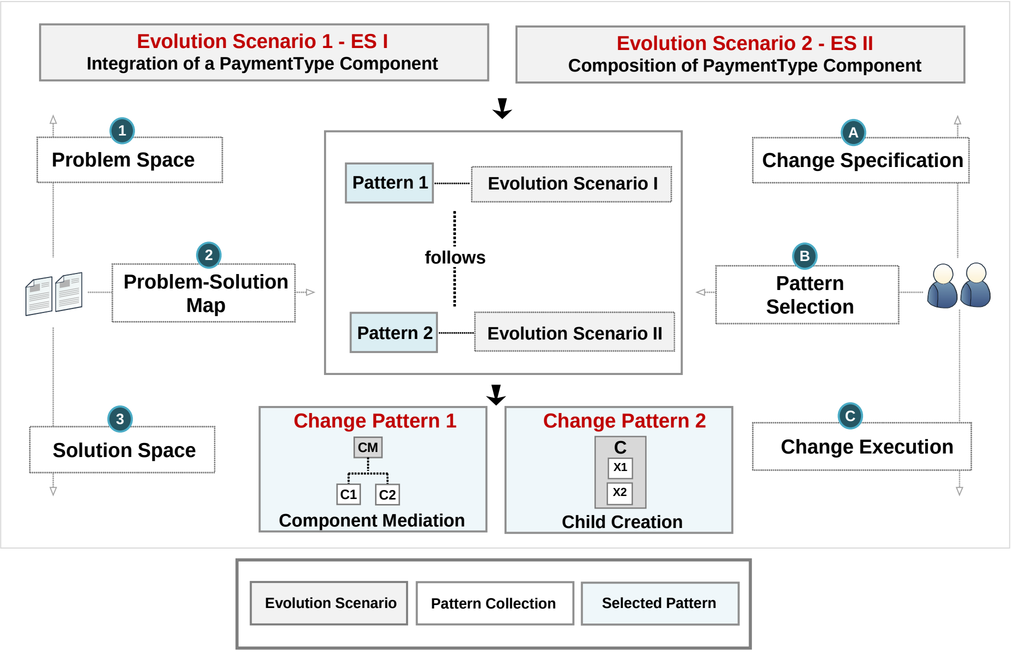


Figure 8.6: An Overview of Mapping Evolution Problems to Available Patterns (Solutions).

As presented in Figure 8.6, in order to enable pattern-driven evolution, we adopt the design space [5] analysis [MacLean 1995, MacLean 1991] to systematically map the problem-solution views to derive a solution. Design space analysis is a methodology to address design-related problems in human-computer-interaction (HCI) [MacLean 1995], however it is generic enough and successfully adopted in the pattern selection context [Zdun 2007]. We utilise design space analysis for an a) explicit representation of alternative change patterns and b) rationale for choosing among

[5] Throughout the chapter, we utilise the terminology ”design space” that refers to a problem and solution space - problems of architecture evolution solved with architecture change patterns.

available patterns (a.k.a. pattern selection). More specifically, Figure 8.6 illustrates:

1. *Problem Space* represents the evolution scenarios, which we identified from the EBPP case study and presented in Table 8.1 (cf. Section 8.4).

2. *Problem-Solution Map* represents the pattern collection that provides a mapping of evolution scenarios to their potential solution as pattern instances in Figure 8.4 (cf. Section 8.3).

3. *Solution Space* represents pattern-driven reuse to guide architecture evolution that is the focus of this section.

8.6.1 Pattern Selection with Design Space Analysis

In a technical context, the generic problem-solution mapping (Figure 8.6) that addresses a given evolution scenario by selecting appropriate pattern(s) for language collection. First we exemplify the Question-Option-Criteria [MacLean 1991] (using design space analysis) that allow us to resolve the pattern selection problem to enable pattern-driven evolution. We illustrate pattern selection in Figure 8.7 by illustrating selection of Component Mediation patterns (cf. Section 8.3 - Pattern 1). Figure 8.7 and the check-list[6] in Table 8.2 that utilise the 3-step QOC-based pattern selection process:

1. *Question - Evolution Scenario* allows the representation of a problem space that allows a declarative specification for intent of change, e.g: *What are the available pattern(s) that allow integration of a mediator component among two (or more) directly connected components*?

2. *Option - Available Patterns* enables problem-solution mapping with selection of the most appropriate pattern from language collection, e.g: The available pattern(s) for component integration is Component Mediation pattern that has two variants Parallel Mediation and Correlated Mediation patterns, in Figure 8.7.

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

[6] Please note that the original check-list for pattern selection is more exhaustive and evaluates all the patterns that exist in the language. For illustrative reasons, we only present a partial check-list based on selection of closely related patterns

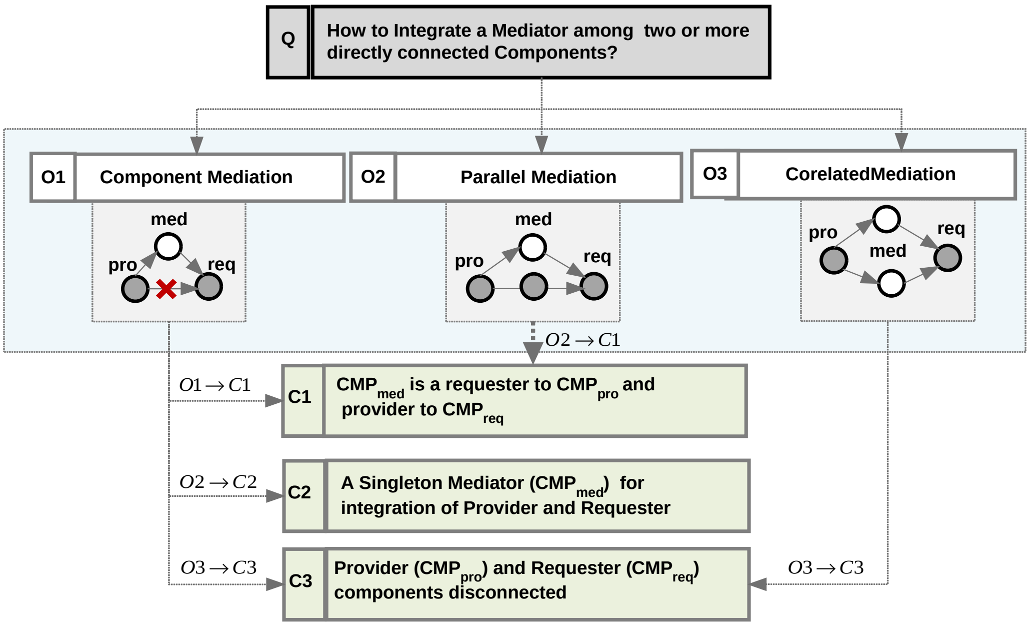

Q
How to Integrate a Mediator among two or more directly connected Components?
O1
Component Mediation
O2
Parallel Mediation
O3
CorelatedMediation
med
pro
req
med
pro
req
pro
req
med
$O2 \rightarrow C1$
$O1 \rightarrow C1$
C1
CMP$_{med}$ is a requester to CMP$_{pro}$ and provider to CMP$_{req}$
$O2 \rightarrow C2$
C2
A Singleton Mediator (CMP$_{med}$) for integration of Provider and Requester
$O3 \rightarrow C3$
C3
Provider (CMP$_{pro}$) and Requester (CMP$_{req}$) components disconnected
$O3 \rightarrow C3$

Figure 8.7: QOC Methodology for Pattern Selection.

Question	*How to Integrate a Mediator among two or more directly connected Components?*		
	Options		
Criteria	Component Mediation	Parallel Mediation	Corelated Mediation
Mediator Addition	✓	✓	✓
Singleton Mediator	✓	✓	×
Component Disconnection	✓	×	×

Table 8.2: QOC Criteria for Selecting an Appropriate Pattern.

8.6.2 Architecture Evolution guided by Change Patterns

After an overview of pattern-driven evolution process and pattern selection, we now focus on architecture evolution that is guided by graph transformation [Baresi 2006a, Loewe 1997]. In the architecture evolution support with a pattern language, we primarily focus on a) enabling *change reuse* and b) maintaining the *structural consistency of architecture* before and after change execution.

We use the Graph Modelling Language (.GML) [Brandes 2002a] for an XML-based representation of architectural instances. This means *specification of architecture models as graph allows us to exploit graph transformation rules to evolve the architecture* in a formal, automated way. More specifically, during execution change operationalisation is abstracted as declarative graph transformation rules (in our case GML transformations). We have already detailed the underlying graph-based formalism for architecture modelling (cf. Section 8.4) and architecture transformation (cf. Section 8.5). It is vital to re-iterate the fact that evolution in the context of composition-based architecture abstracts atomic changes into a set of composite change operations. This means atomic change operations (*Add(), Remove(), Modify()*) on architectural elements (components and connectors) must be abstracted into reusable composite and domain specific changes. Composite-domain specific changes include *Integrate (), Replace (), Decompose (), Split (), Merge ()* etc. of architecture elements.

Architecture Evolution Scenarios

First we present the architecture evolution scenarios, followed by selection and application of change patterns to address the scenarios. In the existing functional scope of the case study (cf. Table 8.1), the company charges its customer with full payment of customer bills in advance of the requested services. Now the company plans to facilitate existing customers with either direct debit or credit-based payments of their bills. In the following, we illustrate the role of Component Mediation followed by Child Creation patterns to allow a) integration of a mediated component PaymentType (ES1) and b) creation of its child components DirectDebit or CreditPayment (ES2).

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

Pattern-based evolution follows a three step process: Change Specification, Pattern Selection and Change Execution, as illustrated in Figure 8.6.

1. *Step I - Change Specification - Questions:* we specify architectural changes along with architectural pre-post-conditions using the GraphML notation [Brandes 2002a]. Change specification essentially provides us with an XML-based notation for intent of the change as detailed in Listing 8.3. A declarative specification allows an architect to represent syntactical context of architectural change that contains the a) source architecture model ($Source < ArchitectureModel >$: Preconditions) b) typed architecture elements ($ArchitectureElement \in ElementType$) that need to be added, removed or modified, and c) anticipated target architecture ($Target < ArchitectureModel >$: Postconditions). A change specification is formally expressed as follows. Please note that specification of change pre-conditions and post-conditions is also a part of the architectural change specification that is detailed below in subsequent steps.

Listing 8.3: A declarative Specification of Architecture Change.

```
<node id = "ChangeRule">
```

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

2. *Step II - Pattern Selection - Options:* in order to select an appropriate pattern, we need to query the pattern language based on pattern-specific conditions. These conditions are expressed as preconditions and post-conditions that must be satisfied to preserve the structural integrity of the overall architecture and individual elements during change execution.

   - The precondition(s) represent the context of architectural elements before change execution in Listing 8.4. For example in Figure 8.8 a, the preconditions (pre) specifies the exact sub-architecture makePayment(BillerCRM, CustPayment) that needs to be changed in the source architecture (source). In order to apply changes, we must find an exact structural match m_s of preconditions in source architecture (cf. DPO transformation [Loewe 1997]), such that $m_s : pre \rightarrow source$ as in Figure 8.8 a. The DPO graph transformation allows a) source architecture (graph) to be transformed into the b) target architecture (graph) by using an intermediate architecture (graph).

Listing 8.4: Preconditions of Architecture Evolution.

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

   - The invariants represent the intermediate architectural structure that is never changed during evolution. This is represented in Figure 8.8 b, the intermediate architecture $m_i : inv \rightarrow intermediate$ with Double-Push-Out (DPO) graph transformations.

   - The post-condition(s) specify the context of evolved architectural elements as a result of the change execution in Listing 8.5. After applying changes on specified elements, the overall architectural structure must be preserved. In order to include the modified architecture

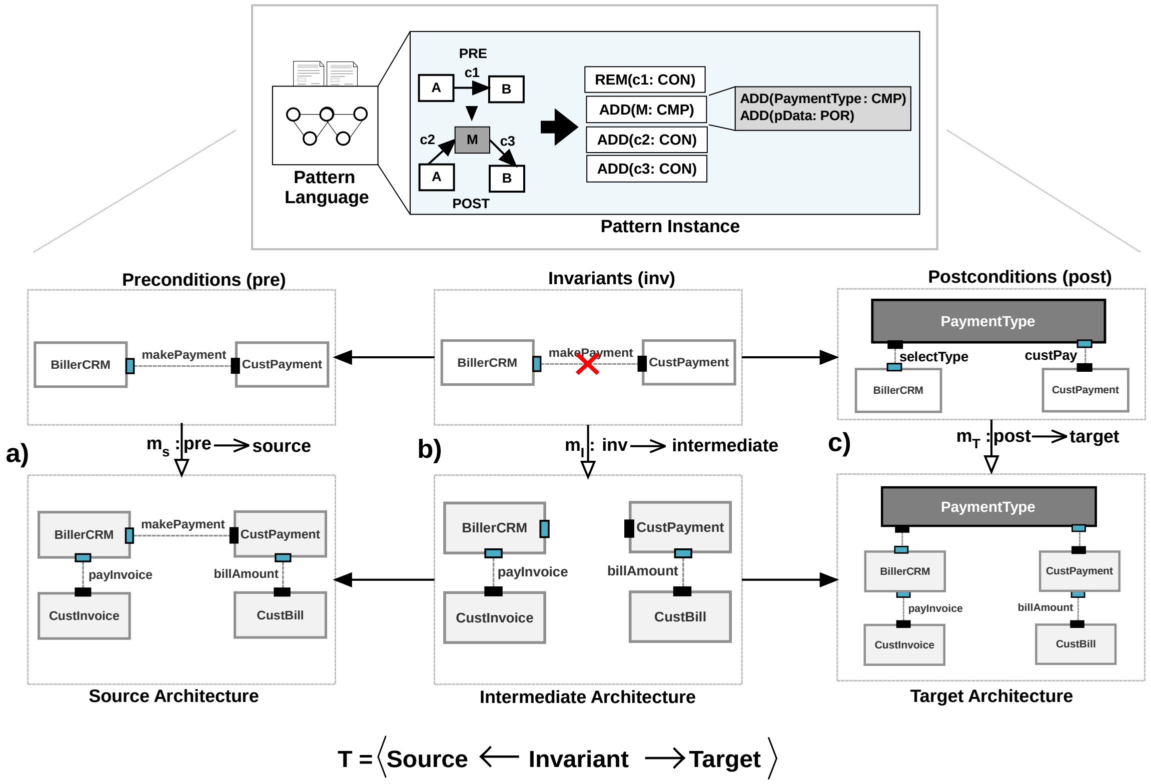


Figure 8.8: Pattern-Driven Architecture Evolution Using Graph-Transformation (DPO) Approach.

elements in the target architecture (target) an exact structural match m_t of post-conditions in the target architecture must exist such that $m_t : post \rightarrow target$ (Figure 8.8) expressed as:

Listing 8.5: Postconditions of Architecture Evolution.

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

3. *Step III - Change Execution - Criteria:* once an exact instance of preconditions in a source architecture is identified, the pattern language is queried with pre-conditions and post-conditions that enables the retrieval of the appropriate pattern that provides the potential

reuse of change operationalisation to enable architectural evolution (cf. Figure 8.7). The query matches the specified change pre-conditions and post-conditions to retrieve the pattern definition. Figure 8.8 illustrates the retrieved instance of Component Mediation pattern. In addition, pattern instantiation involves labeling of generic elements in specifications with labels of concrete architecture elements. For example, in Figure 8.8a the connector instance makePayment that is missing in the change post-conditions is removed from the source architecture. The newly added instance(s) of component PaymentType and connector getType, makePayment are the candidates for addition to the source to obtain target in Figure 8.8c.

Change Operationalisation We provide a brief overview of the change execution that is facilitated using the DPO construction [Loewe 1997, Baresi 2006b] - expressed in Listing 8.6.

- *Pushout I - Deletion Operation* In Figure 8.8b, Source/Intermediate describes the architecture elements to be deleted from the source architecture. For example, the connector makePayment is removed from the BillerCRM and CustPayment. The intermediate architecture is obtained from the source architecture for elements which are a pre-image in Source, but do not exist in Intermediate.

- *Pushout II - Addition Operation* In Figure 8.8c, Target/Intermediate describe the part which needs to be added in Source to obtain Target during change execution. In Figure 8.8c the component PaymentType is added with a connector selectType and custPay in to the architecture. This is represented as:

Listing 8.6: Change Operationalisation for Architecture Evolution.

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

8.7 A Prototype for Pattern-based Architecture Evolution

An overview of the prototype to support pattern-based architecture evolution is presented in Figure 8.9. The list of discovered patterns from Chapter 7 (using the prototype GPride) are provided as an input to the PatEvol prototype.

- The input to this prototype is change rule (specifying the desired changes) along with source architecture model and the pre-conditions and post-conditions for architecture evolution - referred to as *change specification*. Change specification allows the user to declaratively specify the intent of change as the change rule (cf. Section 8.6). The change rule explicitly specify the intent of change, the architecture models to be evolved, i.e., source architecture and pre-conditions on the architecture model. Evolution rules are specified as an XML description (using Graph Modelling Language [Brandes 2002a] notation).

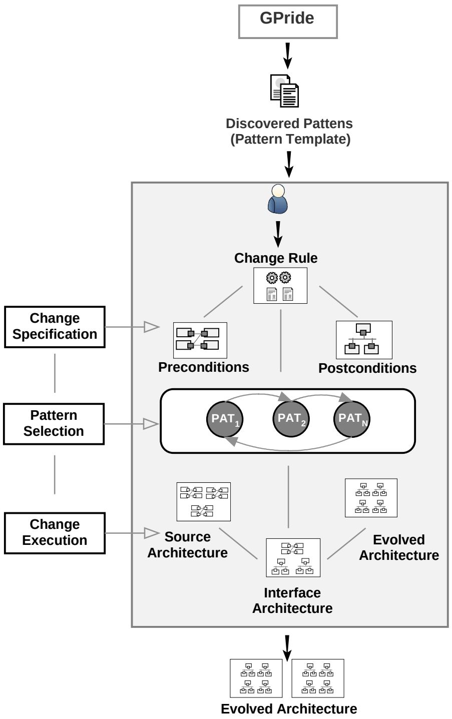


Figure 8.9: Overview of Prototype to Support Pattern-based Architecture Evolution.

- The prototype allows the user to select the most appropriate pattern that can support reuse of architecture change operationalisation - referred to as *pattern selection*. Patterns are expressed in the language as a nested graph (Chapter 7) with GML notation. Pattern selection is enabled with design space analysis based on the QOC methodology [MacLean 1995].

- Finally, the selected pattern guides the architecture evolution with the output as an evolved architecture model - referred to as change execution. Architectural descriptions are provided with a graph-based notation using the graph modelling language (cf. Chapter 2, **Appendix B**). Architectural descriptions before and after evolution are verified with pre-postconditions to ensure structural integrity of the architecture model is preserved during evolution.

Details about the prototype and user interfaces for pattern-based architecture evolution are provided in **Appendix E**.

8.8 Chapter Summary

In this chapter, we aimed at supporting pattern-language based formalism to enable reuse in evolution of component-based software architectures. We highlighted the role of a pattern language as an explicit collection of reuse knowledge to support reuse-driven evolution in CBSAs. More specifically, a pattern language as a system-of-pattern allows problem-solution mapping to reuse change operationalisation that is abstracted using patterns. In a language context, we aim to derive a vocabulary, grammar and pattern sequencing to support pattern application during architecture evolution. The role of pattern language is central in promoting patterns to achieve reuse and consistency in the evolution of CBSAs. We demonstrated that, if an architectural evolution problem can be specified declaratively, pattern-driven evolution could relieve an architect from underlying operational concerns for executing routine evolution tasks facilitated with change patterns.

Chapter 9

Evaluation of the PatEvol Framework

Contents

9.1 Chapter Overview

In this chapter we focus on an experimental evaluation of the architecture change mining and architecture change execution processes in the PatEvol framework. More specifically, in the architecture change mining process, we evaluate the log-based investigation of architecture change representation, change operationalisation and pattern discovery presented in Chapter 5, Chapter 6 and Chapter 7, respectively. In architecture change execution, we evaluate the precision of pattern selection and efficiency of pattern-based architecture evolution from Chapter 8.

9.1.1 Context, Objectives and Methodology of Evaluation

- **Context and Methodology of Evaluation** To evaluate the PatEvol framework, the evaluation methodology is based on ISO/IEC 9126 - 1, an international standard for the evaluation of quality characteristics of a software product and solution [Jung 2004]. However, ISO/IEC 9126 - 1 is a theoretical model for evaluation that needs to be complemented with a concrete evaluation strategy. The evaluation strategy aims to investigate the quality characteristics of PatEvol framework and is based on an experimental investigation with architecture evolution case studies. We engage the participants in case study based experiments and seek their feedback for evaluation of the framework processes.

- **Objectives of Evaluation** The primary objectives of the evaluation are to investigate the quality characteristics of ISO/IEC 9126 - 1 model including *functionality*, *suitability* and *efficiency* of the PatEvol framework. An evaluation of these quality characteristics and their sub-characteristics validate the research hypothesis as outlined in Chapter 1. The hypothesis is further decomposed to identify the research challenges represented as research questions focused on a) RQ1 - *modelling evolution histories* along with b) RQ 2 - *discovery*, c) RQ3 - *selection* and d) RQ 4 - *application* of architecture evolution-reuse knowledge. More specifically:

  1. We evaluate the efficiency and suitability of graph-based modelling of architecture evolution histories (i.e., change logs) - evaluating results corresponding to RQ 1 in Section 9.4.

  2. We evaluate the accuracy and efficiency of pattern discovery algorithms - evaluating results corresponding to RQ 2 in Section 9.5.

3. We evaluate the accuracy of pattern selection process - evaluating results corresponding to RQ 3 in in Section 9.6.

4. We evaluate the reusability and efficiency of pattern application - evaluating results corresponding to RQ 4 in Section 9.7.

Additional details about the *quality sub-characteristics* of ISO/IEC 9126 - 1 model, *experimental setup*, *participants* and the *questionnaire* for participant's feedback to evaluate the framework are presented in **Appendix E**.

9.2 Qualitative Analysis and Comparison of the PatEvol Framework

Before the validation of the *PatEvol* framework, we need to qualitatively analyse and compare the proposed framework with some relevant solutions that enable reuse of architectural evolution. By qualitative comparison we mean the definition and evaluation of the qualitative parameters that allows us to objectively interpret the results. Based on the research state-of-the-art, in Table 9.1 for comparison purposes we have utilised a total of five parameters that include: (i) *Reuse Method*, (ii) *Type of Reuse*, (iii) *Time of Evolution*, (iv) *Architecture Descriptions*, and (v) *Tool support*.

It is vital to mention that, in the software architecture community, pattern oriented software architecture [Buschmann 2007] represents one of the earliest literature on patterns and pattern languages for architecture design. In contrast to patterns of architectural design [Buschmann 2007], our solution (*PatEvol* framework) is the first attempt towards promoting a pattern language to enable reuse in architectural evolution. Table 9.1 provides the basis for a qualitative and comparative analysis (potential and limitations) of our solutions in the context of research state-of-the-art. We are specifically interested to present:

- *What* are the existing approaches that enable reuse-driven evolution in architectures? And *how* are the proposed solutions identical or unique to the existing ones? (See Section 9.2.1)

- *What* is the role of pattern languages in supporting architecture change management? And *why* is there a need for pattern language(s) to evolve software architectures? (See Section 9.2.2)

Solution Reference	**Reuse Method**	**Type of Reuse**	**Time of Evolution**	**Architecture Descriptions**	**Tool Support**
Evolution Styles [Barnes 2014, Le Goaer 2008]	Style-based Reuse	Evolution Plans	Design-time	Component and Connectors	AEvol
Change Patterns [Yskout 2012, Côté 2007]	Pattern-based Reuse	Co-evolution Patterns	Design-time Runtime	Component and Connectors	VIATRA
Pattern Languages [Goedicke 2002, Hentrich 2006]	Pattern Languages	Migration Patterns	Design-time	Object and Service-oriented	MDSD Tool Chain
Proposed Solution	Change Pattern Language	Patterns and Operators	Design-time	Component and Connector	PatEvol

Table 9.1: Comparison of the PatEvol Framework with Research State-of-the-Art.

9.2.1 Reuse-Driven Evolution in Software Architecture

In the context of architecture evolution-reuse knowledge, evolution styles [Barnes 2014, Le Goaer 2008, Tamzalit 2010], and change patterns [Yskout 2012, Côté 2007, Goedicke 2002] emerged as the only notable solutions to enable reuse of design-time as well as run-time evolution of architectures. Evolution styles and change patterns build on the conventional concepts of architecture styles and change patterns to address architectural evolution.

Evolution Styles it is interesting to observe that research in [Barnes 2014, Le Goaer 2008] exploits the same concept (i.e., evolution styles) but address two distinct problems in architecture evolution. More specifically, [Barnes 2014] is the pioneering work on style-driven evolution and is focused on defining, classifying and reusing frequent evolution plans [Barnes 2014]. In contrast to the solution in [Barnes 2014], the authors in [Le Goaer 2008] exploit styles for reusable architecture refactoring. The existing research lacks a consensus about what exactly defines an evolution style, and what is a precise role of evolution styles in architectural change management. In style-driven approaches, notable trends are structural evolution-off-the-shelf and evolution planning with time, cost, and risk analysis to derive evolution plans.

Change Patterns follow reuse-driven methods and techniques to offer a generic solution to frequent evolution problems. In an systematic review[Breivold 2012], the findings highlight that existing solutions overlook the needs for an empirical discovery of evolution patterns. In another systematic reviews of architecture evolution [Jamshidi 2013b], we observed that architecture change patterns [Yskout 2012, Côté 2007] are mostly a proposed solution (based on individual experience) that undermines the fact that patterns represents reuse knowledge that must be empirically discovered.

Based on the comparison in Table 9.1, our solution is fundamentally similar to [Le Goaer 2008] in terms of enabling evolution reuse. However a most notable difference is that instead of inventing styles as architecture evolution reuse knowledge, we propose an empirical discovery of change patterns. During architectural evolution, we also support a (semi-) automated selection of the appropriate change pattern(s) from a collection of the patterns in the language. Considering architecture evolution process, we support a two-step solution for a continuous discovery and application of patterns. In contrast to adaptation or reconfiguration patterns [Yskout 2012, Gomaa 2010] that support run-time evolution, our solution is limited to supporting design-time evolution.

9.2.2 Pattern Languages for Architecture Change Management

Pattern languages provide a formal grammar, vocabulary, and pattern sequencing to derive structure and semantic relationships of patterns in a collection. In the context of architectural change management, the only notable research is on legacy migration [Goedicke 2002] and process-oriented integration [Hentrich 2006] of software architectures. In [Goedicke 2002], the authors propose an incremental migration of legacy software to a flexible architecture using migration patterns. This solution offers a pattern language for migrating C language implementations to components in an object-oriented system.

Based on a comparison in Table 9.1, in contrasts to [Goedicke 2002] our solution is not focused on migration of legacy code to components, instead it supports reuse of architecture evolution. We propose that change patterns as generic reusable abstractions must be empirically *identified as recurring*, *specified once*, and *instantiated multiple times* to benefit evolving architectures.

9.3 Methodology for Evaluating the PatEvol Framework

In the following, first we introduce the ISO/IEC 9126 - 1 standard for quality evaluation of the PatEvol framework in Section 9.3.1 and then present the evaluation strategy in Section 9.3.2. Here, we use the terminologies PatEvol framework or PatEvol or simply framework interchangeably.

9.3.1 ISO/IEC 9126 Model for Quality Evaluation

In 2001 the International Organisation for Standardisation (ISO) published ISO/IEC 9126 - 1 [Jung 2004] an international standard to evaluate quality characteristics of a software product[1]. This standard guides the practical evaluation of a software product or a solution when several stakeholders need to understand, accept and trust the results of evaluation. The ISO/IEC 9126 - 1

[1] Please note that, ISO/IEC 9126 quality model was first issued in year 1991; and later on from 2001 - 2004 ISO issued an international standard (ISO/IEC 9126 - 1) and three technical reports (ISO/IEC 9126 - 2 to ISO/IEC 9126 - 4)

quality model determines six quality characteristics that include *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, and *portability* to evaluate a software product or solution. Furthermore, six quality attributes are sub-divided into a total of 27 sub-characteristics of product quality evaluation. For example, the quality characteristic for **functionality** consists of five sub-characteristics including *suitability*, *accuracy*, *interoperability*, *security*, and *functionality* compliance. For more details about the quality sub-characteristics of ISO/IEC 9126 - 1 model please refer to [Jung 2004].

As presented in Figure 9.1 to evaluate the PatEvol framework, we only consider three quality characteristics of ISO/IEC 9126 - 1 standard *Functionality*, *Efficiency* and *Usability* and their sub-characteristics as illustrated in Figure 9.1.

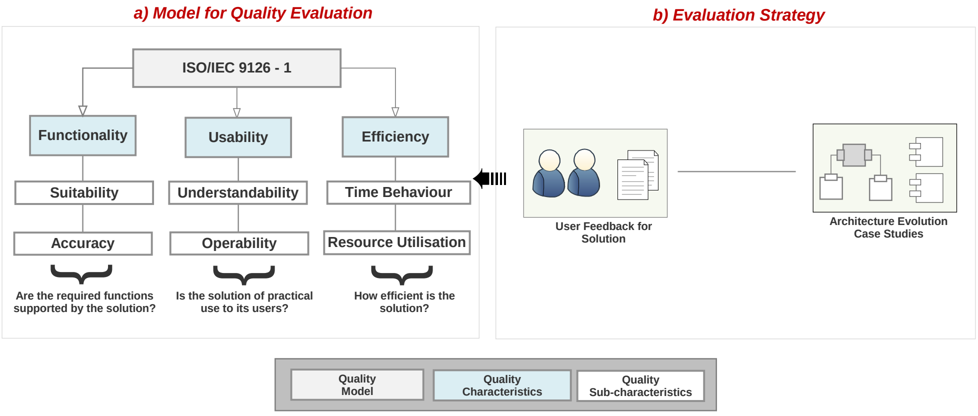

a) Model for Quality Evaluation
b) Evaluation Strategy
ISO/IEC 9126 - 1
Functionality
Usability
Efficiency
Suitability
Understandability
Time Behaviour
Accuracy
Operability
Resource Utilisation
Are the required functions supported by the solution?
Is the solution of practical use to its users?
How efficient is the solution?
User Feedback for Solution
Architecture Evolution Case Studies
Quality Model
Quality Characteristics
Quality Sub-characteristics

Figure 9.1: Overview of the Evaluation for PatEvol Framework.

9.3.2 Evaluation Strategy - Experiments and Participant's Feedback

To evaluate the *functionality*, *usability* and *efficiency* characteristics of the framework; the overall evaluation strategy used in this chapter involves the following two steps also highlighted in Figure 9.1. Details about the experimental set-up for evaluation are provided in **Appendix E, Table E.3**.

- **Experimental Evaluations** We use the evolution case studies and architecture level modifiability analysis (ALMA) [Bengtsson 1999] for an experimental evaluation of the change mining and change execution process in the framework. The ALMA method represents a five step process for a scenario-based evaluation of the architectural modifications and evolution. We prefer ALMA method over other scenario based methods (e.g SAAM [Kazman 1994], ATAM [Kazman 1998]) because ALMA primarily focuses on maintainability aspects of software architectures [Babar 2004] and it can be used at various stages of architectural development to analyse architectural maintenance and evolution. Details of the case studies and ALMA are presented in **Appendix B**. Architecture evolution scenarios from case studies are elicited using the ALMA method. The scenarios provide us an experimental foundation to analyse the sub-quality characteristics of the framework.

- **Evaluation and Participants' Feedback -** we seek usability feedback from five expert participants (detailed in **Appendix E**) for evaluating the functional suitability and performance efficiency of the change mining and change execution processes. More specifically, we engage the framework users in experiments to capture their feedback and to further evaluate the framework. Details about the participants in terms of their affiliation, professional experience and expertise are provided in **Appendix E, Table** E.3.

  Possible Limitations of Usability Analysis - in contrast to the more traditional survey-based research [Clerc 2007, Slyngstad 2008], the participants of the usability evaluation required some basic training to utilise the prototype as well as an introduction and familiarity with the change patterns. Due to some time constraints and the availability of users, we could only engage 5 different users (a.k.a. participants) to evaluate pattern-based evolution. A comparatively small number of participants limits a more comprehensive evaluation as far as capturing the user perspective on pattern-based evolution is concerned. Also, pattern discovery and evaluation of their applicability is a continuous process. The newly discovered patterns must be incrementally validated with their applications on different architectural evolution over time. In this thesis, the discovered patterns from two case studies are cross-validated on a single but different case study. However, to support the generality of the results and their evaluation, based on the finding from [Nagappan 2013] and the guidelines of key informant methodology [Gallivan 2001], we tried to address:

  1. *Sampling of the Participants' Population -* is vital to select the participants (as the representative sample) that had an appropriate knowledge about software design, change implementation and software reuse. The sample had 5 participants with a combined experience of 11 years in software engineering (software design and development related work) with an average experience of more than 2 years per participant. In particular, the participants have a total experience of 8 years with **software architecture** related activities including *architectural design*, *maintenance* and *validation*. The average experience of an individual participant with software architecture related activities is

more than 1.5 years.

2. *Diversity of the Participants in Sample* - ensure participants' skills and knowledge complement the feedback according to different phases of architecture development and evolution. The participants in the sample had expertise in software design (using UML 2.0), software development and testing, and software evolution.

The results based on usability feedback are evaluated in the context of quantitative and qualitative aspects of the ISO/IEC 9126 - 1 quality model.

The developed prototype includes the Graph-based Pattern Identification (*GPride*) to automate pattern discovery from architecture change logs. The input to the prototype *GPride* are architectural changes from logs. The user can customise the pattern discovery process based on specifying the minimum and maximum lengths of pattern candidates and frequency threshold for pattern discovery. The output of the prototype is a list of discovered change patterns. In addition the prototype Pattern-driven Architecture Evolution (*PatEvol*) enables the user/architect to select and apply architecture change patterns for architecture evolution. The input to *PatEvol* prototype is a list of discovered patterns and source architecture model. The output is an evolved architecture model guided by architecture change patterns.

9.4 Evaluating the Efficiency and Suitability of the Log Graph

In Chapter 5, we formalised the change log data as an attributed graph [Ehrig 2004]. Here we aim to evaluate **RQ 1** to analyse if:

- *Graph-based modelling provides a suitable representation of change log data?*
- *In comparison to the more conventional file-based representation of logs, graph-based searching and traversal of log data is efficient?*

This evaluation is also beneficial and a pre-requisite to an efficient graph-based discovery of change patterns from a change log graph. A graph serves as a data structure to model architectural changes and ultimately change pattern mining. An overview of change log data is already provided (in **Appendix C**) to evaluate the sub-characteristics; suitability and efficiency of the ISO/IEC 9126 - 1 quality model. We evaluate the **suitability** of graph-based modelling to represent change log data and the **efficiency** of processing log data that is of significant size (thousands of architectural changes - number expected to grow over time as the data from new logs becomes available).

9.4.1 Suitability of the Change Log Representation

In the context of graph-based pattern discovery, change log graph is a suitable representation as it enables us to exploit sub-graph mining to discover recurrent change sequences as architecture change patterns. In the PatEvol framework graph-based modelling of log data is mandatory, however; in addition to pattern discovery we also need to evaluate if graph provides a suitable representation to analyse and interpret architectural changes. Therefore, we evaluate the suitability of log-based representation based on the feedback by five different participants (Table E.3 in **Appendix E**). We requested these participants to perform the following steps in Table 9.2 to ensure that they are familiar with the concept of change representation in a log as change log graph.

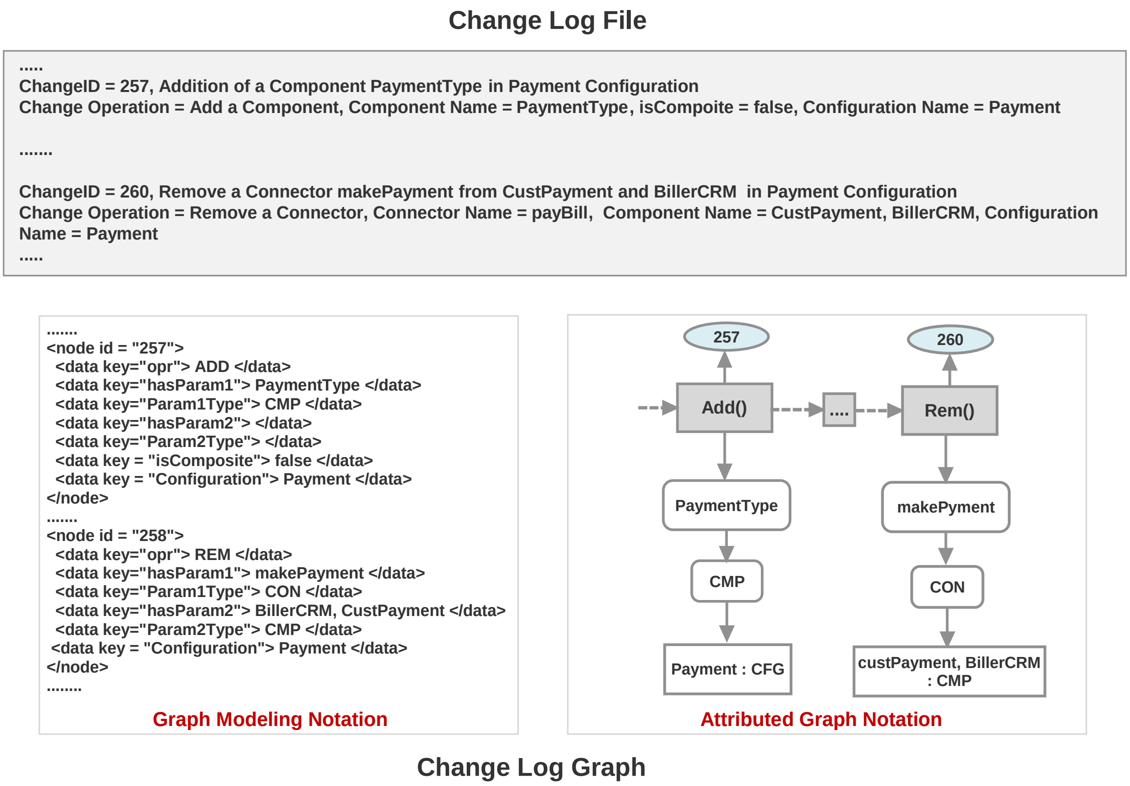
The steps below in Table 9.2 helps the participants to analyse the change log data and also to compare a graph-based representation before we obtain their feedback. The results of this evaluation are gathered by presenting the participants with a questioner to capture their feedback as presented in Table 9.3. For example, when asked about:

Step	Action
Step I	Analyse a sample **change log file** that contains a total of 200 atomic change operations (from Table E.3, Appendix E) as the sample log graph presented in Figure 9.2
Step II	Analyse the sample **change log graph** that consists of 200 nodes as change operations (created from the log file) in Figure 9.2
Step III	Identify a change operation (both in the log file and log graph) that enables addition of an architectural component PaymentType
Step IV	Convert at-least 3 change operations from change log file into a change log graph.
Step V	Addition of change operations in the change log file and repeat Step IV. **A -** We requested the users to add a new configuration, a component containing a port in the configuration as an entry of 3 change operations in the log file. **B -** The users then represented these change operations in the log graph file.

Table 9.2: A Summary of Step and Actions by Participants for Evaluating Log Graph Suitability.

- **Question** Which of the two provides a suitable representation of change operationalisation on architecture elements?

- **Response** The participants provided their preference as either a change log file, change log graph or mention if they are not sure about any of the two. In addition the participants can also provide some comments or additional details about their feedback.



Change Log File

```
.....
ChangeID = 257, Addition of a Component PaymentType in Payment Configuration
Change Operation = Add a Component, Component Name = PaymentType, isCompoite = false, Configuration Name = Payment

.......

ChangeID = 260, Remove a Connector makePayment from CustPayment and BillerCRM  in Payment Configuration
Change Operation = Remove a Connector, Connector Name = payBill,  Component Name = CustPayment, BillerCRM, Configuration
Name = Payment
.....
```

```
.......
<node id = "257">
  <data key="opr"> ADD </data>
  <data key="hasParam1"> PaymentType </data>
  <data key="Param1Type"> CMP </data>
  <data key="hasParam2"> </data>
  <data key="Param2Type"> </data>
  <data key = "isComposite"> false </data>
  <data key = "Configuration"> Payment </data>
</node>
.......
<node id = "258">
  <data key="opr"> REM </data>
  <data key="hasParam1"> makePayment </data>
  <data key="Param1Type"> CON </data>
  <data key="hasParam2"> BillerCRM, CustPayment </data>
  <data key="Param2Type"> CMP </data>
 <data key = "Configuration"> Payment </data>
</node>
........
```

Figure 9.2: Comparison Overview of Log Graph and Log File.

In Table 9.3, we capture the participant's feedback to analyse the suitability of representation, interpretation, visualisation, searching/retrieval and addition of architecture change operations. Additional details about the questionnaire presented to the participants are provided in **Appendix E**.

Please tick the most appropriate option. Also provide comments, if required					
Questions from Participants		**User Feedback**			
Which of the two provides a/an		Log File	Log Graph	Not Sure	Any Comments
Q1	Suitable representation of change operationalisation on architecture elements				
Q2	Easy interpret of the intent of architecture change operations				
Q3	Visualisation of changes on architecture elements				
Q4	Easy to search and retrieve the log data				
Q5	Easy to record change operations				

Table 9.3: Questionnaire for Participant's Feedback on Suitability of Log Graph.

9.4.2 Summary of the User Feedback for Suitability of Change Log Graph

We provide a summary of the feedback as the evaluation results and highlights our key findings as below. Details about the experimental feedback and more specifically participants for the framework evaluation are provided in **Appendix E (Table E.3)**.

- Based on the feedback a total of 3 out of 5 participants agreed that change log graph provides a suitable representation and easy interpretation of log data as no key information about individual change operation is omitted when modelling log data as a graph.

> *When considering a single change operation at a time there is no such difference between the interpretation of a log file and change log graph. However, analysing a sequence of changes - as interconnected nodes - it is easy to understand nodes as operations than a collection of tuples in the log file.*

- A graph representation (operation as an individual node) helps the user to understand a collection or sequence of changes more easily.
- It is agreed among all the participants that when a change log is visualised as a graph (Figure 9.2) it is intuitive to see change operations as nodes and directed edges as a sequence among the change operations. Therefore it is easy to interpret the intent of change operationalisation and their sequences.

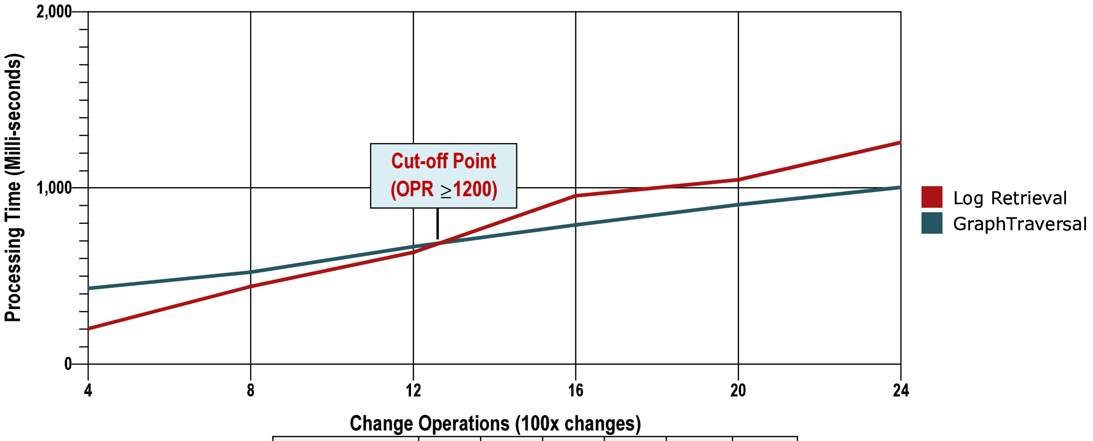
> *One of the participants was not sure about any significant distinction between data representation of change log file vs change log graph. The participant disagreed about the suitability of change log graph as it requires an extra effort to create the change log graph that is time consuming and error prone as a manual effort.*

- In terms of a manual effort to search and retrieve log data the participant's feedback do not suggest any significance of change log graph. All the users agreed that recording change operations as a graph is more trivial and easy than recording changes in a change log file.

9.4.3 Efficiency of the Graph-based Retrieval of Log Data

Once we have evaluated the suitability of a change log graph, we now focus on analysing the efficiency of graph-based retrieval of the log data. The primary objective of this evaluation is to analyse the efficiency of graph-based investigation in comparison to the traditional file-based system for searching and retrieving log data of significant size.

In general, graph-based efficiency in the context of log data refers to a) efficient representation and b) efficient retrieval of change operations. Here we only focus on time-efficiency, i.e., time (T in milliseconds (ms): Y-axis) to search and retrieve change instances (per 100 change operations: X-axis) for traditional file-based retrieval vs graph traversal in Figure 9.3. In Figure 9.3, we illustrate a relative comparison of the time taken to retrieval log data (log data for evaluation provided in **Appendix E**, Table E.2) from change log file at an interval of 400 change operations.



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

Figure 9.3: Comparative Analysis of Time Taken (Log-based Retrieval vs Graph-based Traversal).

9.4.4 Summary of Results for Efficiency of Log-based Data Retrieval

We now present a summary of the evaluation of the efficiency of retrieving data from change log graph.

- *Pre-processing of Log Data for Pattern Discovery* - in order to enable graph-based modelling, additional overhead (on average about **400 ms**) involves creation of change log also considered as a pre-processing for graph-based pattern discovery. In this pre-processing, the change operations and their sequence from log file are mapped to their corresponding nodes and edges in change log graph.

- *Graph vs File-based Traversal of Log Data* - as the size of the log data increases, graph-based traversal starts to outperform file-based retrieval.

*Whenever the number of change operation being queried increases (on average) by more than **1200**, graph-traversal and retrieval is always time efficient. We conclude that although graph-based modelling involves additional pre-processing, however, **we found a cut-off point at Opr > 1200**, where graph traversal outperforms file-based retrieval.*

We evaluate the solution in the context of RQ 1 that aimed to address a suitable modelling notation of change log data for an experimental discovery of architecture evolution knowledge. Based on the evaluations, we generalise the measure of the efficiency (**E**) of the change log graph traversal in terms of time it takes (**T**) to analyse total graph nodes (**N**). **T** and **N** are proportional, whereas ϕ represents a constant time required to create the change log graph (approximately 400 milliseconds).

$$E = N/T + \phi$$

The relation between the time taken to traverse the graph generalises the findings (cf. Figure 9.3) that as the number of graph nodes increases, there is a minor increase of time to traverse them. For example, when the number of nodes grow from 1200 to 2400 (doubled), there is only an increase of less then 350 milliseconds approx in Figure 9.3.

A log graph provides a suitable representation and efficient manipulation of log data. However, to achieve this an initial effort is required to map each individual change from log file to an individual node in the log graph. It requires a tool support to create change log graph from log file. Once log graph is created, it provides a faster and efficient data structure for searching and retrieval of log data. Log graph also provides the foundation for graph-based mining of patterns from change logs.

9.5 Evaluating Accuracy and Efficiency of Pattern Discovery

In Chapter 6 and Chapter 7 we exploited the graph-based formalism to classify change operationlisation and to discover architecture change patterns using change log graph. Here we aim to evaluate **RQ 2** to analyse if

- *Pattern discovery algorithms provide (an automated) solution that ensures accuracy of pattern discovery from change logs?*

- *In contrast to manual discovery, the algorithms provide an efficient solution to discover architecture change patterns from logs?*

We identified architecture evolution scenarios (using the ALMA method in **Appendix E**). These evolution scenario are used for an experimental evaluation of pattern discovery algorithms. In the remainder of this section, we evaluate the sub-characteristics *accuracy* and *efficiency* of the ISO/IEC 9126 - 1 quality model, to analyse the performance of pattern discovery algorithms. Algorithmic performance is measured in terms of the accuracy (correctness and completeness) and efficiency (time efficiency) of pattern discovery.

We evaluate the accuracy and efficiency of the change pattern discovery algorithms by means of a comparison between the results of algorithms and the outcome of a manual approach for pattern discovery. The manual discovery is performed by five different participants. We provide the participants with a) a sample log file and b) some evolution scenarios along with a verbal description of the file and the evolution scenarios. Once the user agree that they had a clear idea about the pattern discovery process, we ask them to discover recurring change sequences as patterns and compare the results with discovery algorithms.

9.5.1 Interpretation of Results - Automated vs Manual Discovery of Patterns

We now discuss the results of comparison between manual and automated pattern discovery that is summarised in Table 9.4. The comparison is based on six different comparison criteria (C1 - C6) in Table 9.4. Four comparison criteria C1 - C4 represent a relative measure of the accuracy of the pattern discovery mechanism. In addition two criteria C5 - C6 represent the relative measure of efficiency in Table 9.4.

Comparison of the Accuracy of Pattern Discovery

1. *Discovered Pattern Instances -* It provides a comparison of the relative accuracy in terms of discovering the existing patterns in the sample log file. The comparison confirms the accuracy of proposed algorithms in terms of the completeness and correctness of our solution.

2. We requested the participants to cross-verify the discovered change patterns to confirm that algorithms discovered the patterns that were omitted during manual discovery.

ID	Comparison Criteria	Manual Discovery Users	Automated Discovery Algorithms
N/A	Log Size (Total Change Operations)	212 Atomic Change Operations	

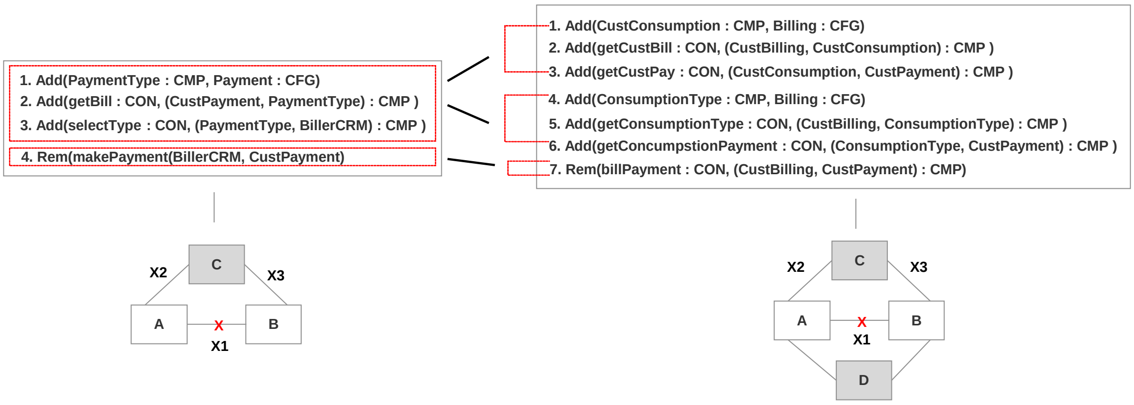
Accuracy Comparisons			
C1	Discovered Pattern Instances	2	4
C2	Overlapping Patterns	2	4
C3	Inexact : Inexact Instances	2 : 0	2 : 2
C4	Pattern Discovery Precision	0.5	1.0

Efficiency Comparisons			
C5	Time Taken	> 25 minute	< 6 second
C6	Candidate Identification Required	Yes	Yes

Table 9.4: Summary of Comparison between Manual vs. Automated Pattern Discovery.

> *No change pattern is omitted by the algorithm. In contrast, while following a manual approach we were able to discover only two change patterns where the algorithm discovered four change patterns as presented in Table 9.4.*

3. *Discovery of Overlapping Patterns* - The experimental analysis suggests that manual discovery do not support discovery of an overlapping pattern. An overlapping pattern is also referred to as a partial pattern (cf. partial exact sequence in Chapter 6) as it overlaps (with a part of) another pattern.


1. Add(PaymentType : CMP, Payment : CFG)
2. Add(getBill : CON, (CustPayment, PaymentType) : CMP)
3. Add(selectType : CON, (PaymentType, BillerCRM) : CMP)
4. Rem(makePayment(BillerCRM, CustPayment)
1. Add(CustConsumption : CMP, Billing : CFG)
2. Add(getCustBill : CON, (CustBilling, CustConsumption) : CMP)
3. Add(getCustPay : CON, (CustConsumption, CustPayment) : CMP)
4. Add(ConsumptionType : CMP, Billing : CFG)
5. Add(getConsumptionType : CON, (CustBilling, ConsumptionType) : CMP)
6. Add(getConcumpstionPayment : CON, (ConsumptionType, CustPayment) : CMP)
7. Rem(billPayment : CON, (CustBilling, CustPayment) : CMP)
X2
C
X3
A
X
B
X1
X2
C
X3
A
X
B
X1
D

a. Pattern I - Omitted With Manual Analysis

b. Pattern II - Discovered with Manual Analysis

Figure 9.4: Overview of the Pattern Overlap.

For example, in Figure 9.4 the change operations in Pattern I overlaps with change operations in Pattern II - we refer to this as Pattern I overlaps Pattern II or Pattern I is a part of Pattern II.

- With a manual discovery, the overlapping patterns are not discovered. The primary reason to omit an overlapping pattern is that once a pattern has been discovered, manual analysis did not considered analysing the recurrent change operations in that pattern.

- The process to discover partial exact and partial inexact sequences is already detailed in Chapter 6. For example Pattern I in Figure 9.4 a is an instance of Component Mediation pattern with its variant Pattern II represents Parallel Mediation pattern that are detailed in Chapter 7 and Chapter 8.

> *The evaluations suggests that manual analysis did not consider that part of a pattern may also be a pattern. In contrast, the pattern discovery algorithms are able to discover the overlapping patterns that represent a pattern variant*

4. *Discovery of Exact and Inexact Sequences* - The accuracy of pattern discovery algorithms is enhanced with their ability to identify the exact as well as an inexact instance of change patterns. As we detailed in Chapter 6, it is quite common that in a change log that two or more change sequences may have a distinct order of change operations but their impact of change is identical.

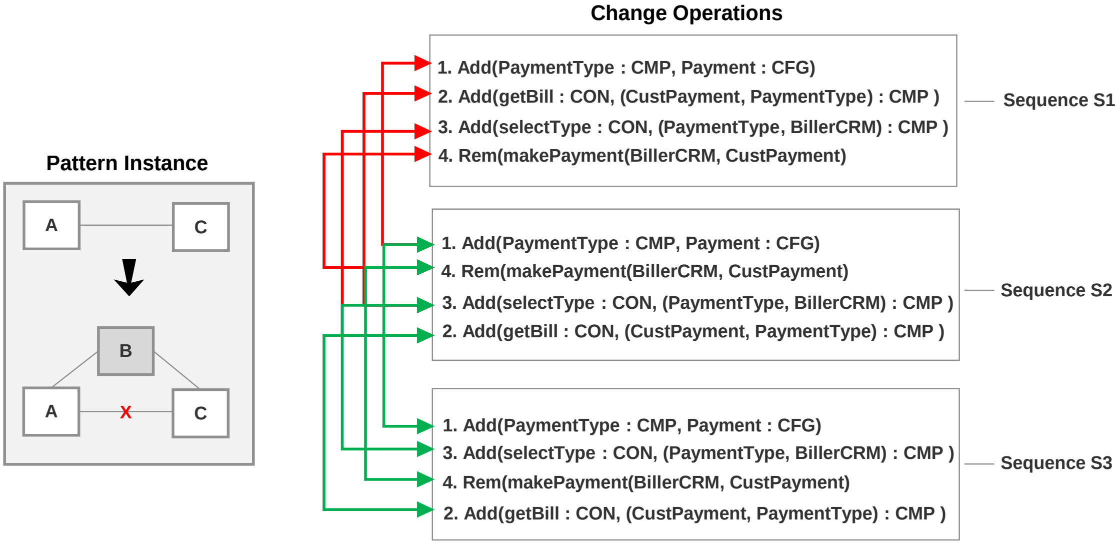

Change Operations
Pattern Instance
A
C
B
A
X
C
1. Add(PaymentType : CMP, Payment : CFG)
2. Add(getBill : CON, (CustPayment, PaymentType) : CMP)
3. Add(selectType : CON, (PaymentType, BillerCRM) : CMP)
4. Rem(makePayment(BillerCRM, CustPayment)
Sequence S1
1. Add(PaymentType : CMP, Payment : CFG)
4. Rem(makePayment(BillerCRM, CustPayment)
3. Add(selectType : CON, (PaymentType, BillerCRM) : CMP)
2. Add(getBill : CON, (CustPayment, PaymentType) : CMP)
Sequence S2
1. Add(PaymentType : CMP, Payment : CFG)
3. Add(selectType : CON, (PaymentType, BillerCRM) : CMP)
4. Rem(makePayment(BillerCRM, CustPayment)
2. Add(getBill : CON, (CustPayment, PaymentType) : CMP)
Sequence S3

Figure 9.5: Ordering of the Operations in Change Patterns.

For example, in Figure 9.5 the order of change operations in Component Mediation pattern is represented as three distinct sequences *S1*, *S2*, and *S3*. The order of change operations is expressed as: $S_1 = \{OP1, OPR2, OPR3, OPR4\}$ that can also be represented as $\{OPR1, OPR4, OPR3, OPR2\}$ or $\{OPR1, OPR3, OPR4, OPR2\}$ and so on (commutative change operations). For illustrative purposes the Figure 9.5 only represents a minimal example where the number of change operations in four, however as the number of change operations in a sequence grows the number of sequences with different orders also grow.

Manual discovery is error prone in discovering the inexact matches. In contrast the pattern discovery algorithms in Chapter 7 enables discovery of exact as well as inexact sequences as change patterns.

5. *Evaluating the Precision of Pattern Discovery Algorithms* - We now discuss the results of evaluation for pattern discovery precision. Pattern discovery precision refers to the accuracy of the discovery algorithms as a ratio of the discovered patterns to existing patterns. We formally define the pattern discovery precision as:

   Pattern Discovery Precision (P) *refers to the total number of patterns discovered from log divided by the total number of patterns that exist in the log expressed as*:

$$\frac{\mid Discovered\ Patterns\ from\ Log \mid}{\mid Existing\ Patterns\ in\ Log \mid}$$

   For example, Figure 9.6 represents the entire pattern space or ultimately the discovery space (all changes that exist in the log), existing change patterns are represented as **C**.

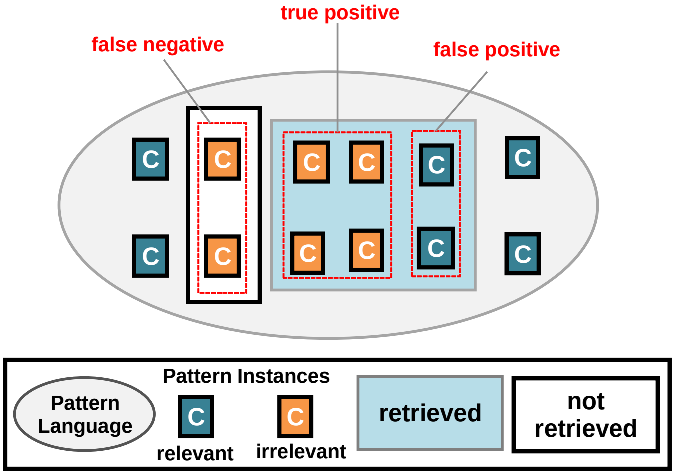


Figure 9.6: Overview of the Pattern Selection Precision and Recall.

The log contains a collection of valid and invalid patterns (detailed in Chapter 7, **Algorithm II Candidate Validation**) - during the discovery process, we have three scenarios.

- The discovered patterns are relevant (a.k.a. true positive)

- The discovered patterns are irrelevant (a.k.a. false positive)

- The relevant patterns have not been discovered (a.k.a. false negative)

> *The precision is a relative measure of accuracy of the solution to discover patterns - discovering the exact as well as inexact pattern instances. Based on our experiments, the algorithms have a precision of* ***1.0*** *while the manual solution represents a precision of* ***0.5****.*

Comparison of the Efficiency of Pattern Discovery

6. *Processing Time for Pattern Discovery* It refers to the time taken to discovery the change patterns. This result shows that manual discovery is a time-consuming process, whereas our algorithms are accurate and time-efficient for pattern discovery.

> *Based on the comparison in Table 9.4, the users took more than 25 minutes to go through a small subset of the change log and to discover the change patterns in comparison to the algorithms that took less than a few seconds (log size 200 change operations).*

7. *Candidate Identification* Another comparison is the identification of pattern candidates. A pattern candidate represent a potential pattern depending on its occurrence frequency in the logs. The manual discovery process requires that pattern candidates must be identified. The algorithms also rely on candidates for pattern discovery that is considered to be the pre-processing for pattern discovery (Chapter 7, **Algorithm I Candidate Generation**).

9.5.2 Discussion and Conclusions

Discovery of change patterns (in Chapter 7) not only helps in identifying the frequent architectural changes, but also reduces any manual effort in terms of process accuracy and efficiency. Moreover, the pattern discovery algorithms also help us to eliminate the false positive patterns that may violate the structural integrity of an architecture model. In this section, we have evaluated the discovery algorithms by a comparison with the manual approach in terms of its correctness and completeness of the algorithmic functionality. Our experimental analysis suggests that:

There does not exist any solution(s) for pattern discovery from logs that provides us with a benchmark evaluations. In comparison to a manual approach the automated approach for architecture change pattern discovery is beneficial in different ways. More specifically, a manual discovery of change patterns is impractical when the size of log data is significant. As the size of change log data increases, the processing time required for manually discovery of patterns increase significantly - automated approach becomes inevitable.

While the results of the algorithms were accurate, the manual approach failed to manually identify the inexact pattern instances. The output of the algorithm was verified by taking participants feedback (in Table 9.4 and details in **Appendix E**) to confirm the correctness and the completeness of the algorithm.

9.6 Evaluating the Accuracy of Pattern Selection

In Chapter 8, we presented the Question Option Criteria (QOC) [MacLean 1991] methodology to enable a systematic selection of architecture change patterns. Here we aim to evaluate **RQ 3**.

- *Does the solution enables accuracy of the pattern selection from change pattern language?*

In the remainder of this section we evaluate the sub-characteristics accuracy of the ISO/IEC 9126 - 1 quality model to evaluate pattern selection process. The accuracy of the pattern selection is evaluated based on analysing the precision and recall of selection process.

9.6.1 Accuracy of Pattern Selection - Precision and Recall Measure

After evaluating the accuracy and efficiency of pattern discovery, we now focus on analysing the accuracy of pattern selection. Pattern selection is a critical challenge, especially when the number of change patterns is large in a pattern collection or the collection is expected to grow overtime. Moreover, even if we consider a limited number of patterns; still a pattern user must be aware of all the existing patterns and must understand the internal structure of a pattern collection to select the most appropriate patterns.

In a similar problem of pattern selection in [Kampffmeyer 2007], the solution assists the users with selection of design patterns by formalising the intent of 23 patterns from Gang-of-Four (GOF) pattern collection [Gamma 2001]. Based on the formalisation of the patterns intent, the solution offers a design pattern wizard that enables a user to follow a step-wise process to select the applicable design patterns based on a description of a design problem. Our solution and its evaluation are fundamentally different to the research in [Kampffmeyer 2007]. We must enable a user to select the most appropriate patterns from a collection that continuously evolves (number of patterns is expected to increase rather than a fixed collection). Furthermore, *GOF patterns are well known in the software developer's community, whereas the proposed change patterns are a new concept with a lack of understanding for the first time users*. To ensure that inexperienced users or architects are able to select appropriate patterns, we must evaluate the accuracy of pattern selection that requires a precise mapping of the evolution problems specified by the user and the solution represented as available pattern(s). We evaluate the precision and recall factor for change pattern selection.

- **Measuring Pattern Selection Precision (P)** - *is defined as number of relevant pattern instances retrieved by a search divided by the total number of pattern instances retrieved*. It is expressed as:

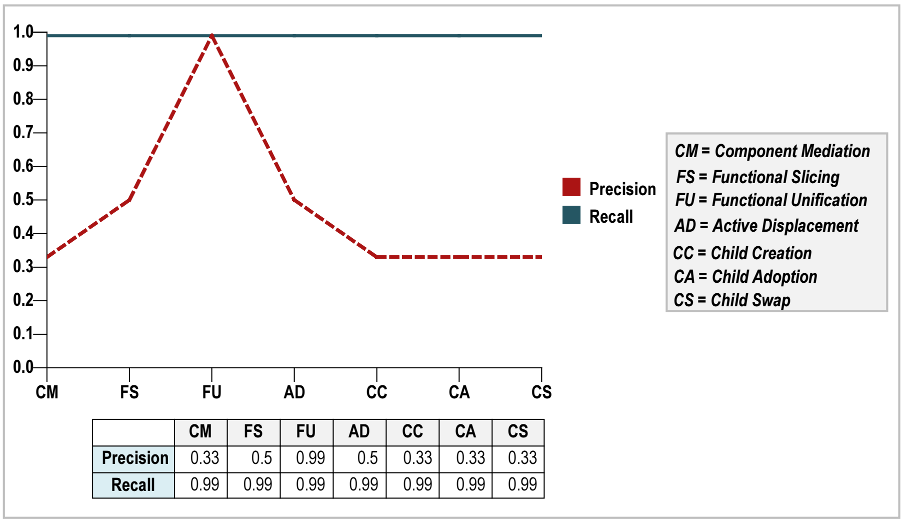
$$\frac{\mid Relevant\ \ Patterns\ \ Retrieved \mid}{\mid Total\ \ Patterns\ \ Retrieved \mid}$$

- **Measuring Pattern Selection Recall (R)** - *is defined as number of relevant pattern instances retrieved by a search divided by the total number of existing relevant pattern*. It is expressed as:

$$\frac{\mid Relevant\ \ Patterns\ \ Retrieved \mid}{\mid Total\ \ Existing\ \ Relevant\ \ Patterns \mid}$$

A summary graph of precision and recall is presented in Figure 9.7. Based on the precision and recall criteria above, the results of pattern selection recall results in a high value indicating that the solution is able to retrieve approximately all of the relevant patterns from the language. Please note that due to a smaller search space (*7 patterns and 2 variants*) the recall is measured to be 0.99 approx. for all pattern instances. We believe that a high recall is a results of the smaller search space for the patterns. We can only objectively evaluate the results of pattern selection recall when we have a larger pattern space - evaluating and comparing selection recall with manual selection and QOC method. This means that a smaller pattern space represents a threat to the validity of recall results whenever the size (total number of patterns in) pattern space grows overtime.

A high recall suggests the solution is adequate in selecting the most relevant instances from available collection. However, we experience a different behaviour for precision because identification of the exact pattern in the context of related patterns is more challenging.



	CM	FS	FU	AD	CC	CA	CS
Precision	0.33	0.5	0.99	0.5	0.33	0.33	0.33
Recall	0.99	0.99	0.99	0.99	0.99	0.99	0.99

Figure 9.7: Precision and Recall for Pattern Selection.

The corresponding values for selection precision varies between 0.33 and 0.99. Whenever we query for ”component integration pattern”, we are returned with at least three pattern instances (Component Mediation, Parallel Mediation, and Correlated Mediation).

A relative low precision suggests that an improvement of the pattern selection is required. Even with a small search space (7+2) patterns the solution is not very accurate in selecting the most appropriate patterns.

To improve the selection precision, we classify the architecture change patterns and then re-evaluate the results for pattern selection.

9.6.2 Effects of Pattern Classification on Selection Precision

The classification of change patterns in Figure 9.8 helps us to organise the patterns into a group of functionally related patterns. More specifically, it enables a logical grouping of related patterns based on the types of architectural changes that a group of patterns support. We have classified the existing patterns into three distinct types as *Composition*, *Association* and *Decomposition* patterns with classification ids 1, 2, and 3 respectively, presented in Figure 9.8. For example, in Figure 9.8 *the* Child Creation *pattern enables the composition of an atomic component into a composite one that contains one or more child components and is classified as* Composition Type *pattern*. In the pattern language context, the classification has no effect on pattern relations - pattern(s) in one classification may be related to pattern(s) in a different classification.

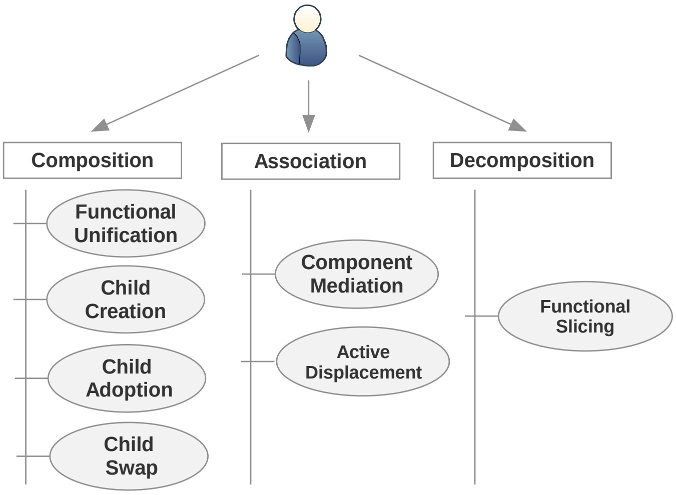
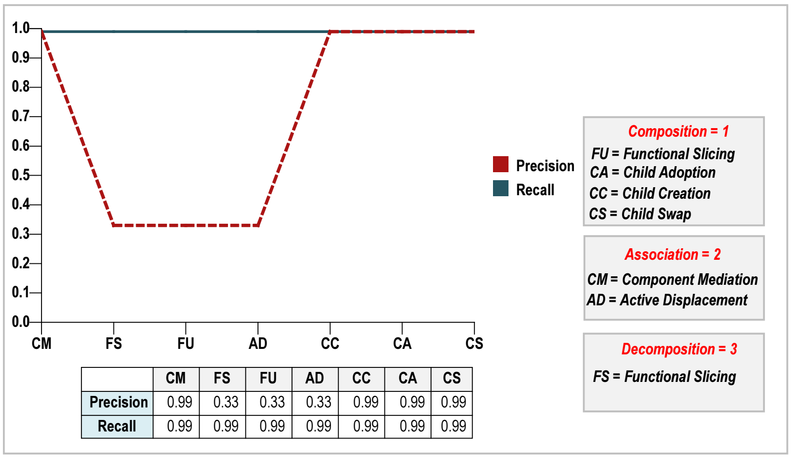


Figure 9.8: An Overview of Change Pattern Classification.

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

A possible limitation for pattern type classification is that the user must specify the classification type in which they aim to search the patterns. If the type is not specified correctly, the appropriate pattern(s) must be searched in all the classifications which results in a lower precision - as illustrated in Figure 9.7.

Based on the pattern classification in Figure 9.8, we again evaluate the precision and recall of pattern selection. A summary graph of the precision and recall is presented in Figure 9.9. Pattern classification has no impact on selection recall factor that remains as 0.99 also highlighted in Figure 9.7. However, we are able to increase the selection precision factor because the pattern search space is minimised with the pattern type classification. The selection precision for association and decomposition type patterns is 0.99. The precision for composition type pattern is 0.5 - a low precision is a consequence of the overlap of change support by Child Swap, Child Creation and Child Adoption patterns is 0.33 that is subject to further evaluations.



	CM	FS	FU	AD	CC	CA	CS
Precision	0.99	0.33	0.33	0.33	0.99	0.99	0.99
Recall	0.99	0.99	0.99	0.99	0.99	0.99	0.99

Figure 9.9: Accuracy of Pattern Selection based on Pattern Classification.

9.6.3 Implications of a Small Search Space for Pattern Selection Precision

The current search space (total patterns in the language) represents a relatively small number (7+2) patterns that limits a strong judgement or claims about pattern selection precision. A small search space is a consequence of change patterns being a new concept when compared to the more established GOF design patterns [Gamma 2001] and their selection [Kampffmeyer 2007]. Moreover, an empirical discovery of new change patterns requires the availability of more data in terms of investigating architecture evolution histories that represent a possible dimension of the future research.

Based on pattern selection research in [Zdun 2007, Kampffmeyer 2007], the small search space represents a possible threat to the validity and generalisation of pattern selection precision (cf. Figure 9.9). It also raises the following concerns:

1. What is the measure of selection precision when the number of patterns in the language grows?
2. Is there an increase in the complexity and total time taken for pattern selection?
3. Does the user need more knowledge about existing change patterns for an accurate selection?

Currently the small pattern search space can only be compensated in the future with the addition of new patterns in the language over-time. However, as the number of patterns grow in the language the complexity and the effort for pattern selection increases [Zdun 2007]. In order to tackle this challenge, we have utilised the QOC methodology [MacLean 1995] as a systematic and semi-automated approach for pattern selection (already detailed in Chapter 8). Specifically, in comparison to any manual methods (that are error-prone and time consuming) [Kampffmeyer 2007] for pattern selection the QOC method supports a semi-automated approach âĂŞ supporting the necessary user intervention for pattern selection from the language. By exploiting the QOC methodology, the user can query for the appropriate pattern (if exists) based on the predefined criteria.

The current evaluations suggests that even when the number of patterns may grow, the user can select the appropriate patterns with minimal or sometimes no knowledge about the individual patterns by following a systematic approach. However, the impact of the increased number of patterns on pattern selection, complexity and time taken can only be objectively evaluated when new patterns are incorporated by investigating new data (as it becomes available) from different logs.

9.7 Evaluating the Efficiency and Reusability of Pattern-based Architecture Evolution

In Chapter 8, we presented pattern-based evolution to support reusability and consistency of architectural changes. Here, we aim to evaluate **RQ 4** to analyse if:

- *Does the application of change patterns to evolve architectures enhances the efficiency of the architecture evolution process?*
- *Do the change patterns enable reuse of change operations for architectural change implementation?*

In the remainder of this section, we evaluate the sub-characteristics efficiency and reusability to evaluate pattern-based architecture evolution process. The **efficiency** of architecture evolution process is evaluated in terms of number of change operations required and the time taken to implement the required architectural changes.

The ISO/IEC 9126 - 1 model do not consider reusability as a sub-characteristics of software product quality. In our evaluation, we need to evaluate the **reusability** of architectural changes. Therefore, we borrow reusability - a sub-characteristics of maintainability - from ISO/IEC 25010 standard. In ISO/IEC 25010 standard, *"reusability refers to the degree to which an asset can be used in more than one software system, or in building other asset* [ISO/IEC25010 2010]. We utilise the case of architectural evolution of a peer-to-peer appointment system to evaluate patter-based evolution.

Reusabiliy of change has an impact on the **granularity of architectural change**. The granularity of architectural changes refers to changes applied to different levels or different architecture elements in the architecture model [Buckley 2005]. In terms of granularity, architectural changes can be classified as coarse-grained and fine-grained changes. In the context of CBSAs, coarse-grained changes include addition, removal and modification of architectural components and connectors only. In contrast, the fine-grained changes include addition, removal and modification of architectural components, their ports, connectors as well as their endpoints (cf. Chapter 6). We evaluate the effects of reusability on granularity of architectural changes.

9.7.1 Pattern-based Evolution of a Peer-to-Peer Appointment System to Client-Server Architecture

A high-level architectural view of the peer-to-peer appointment system (P2P-AS) [Rosa 2004] is presented in Figure 9.10. Architectural components and connectors are represented inside configurations for modelling of P2P-AS system. The pattern discovered from EBPP [EBPPCaseStudy] and 3-in-1 Telephone System [3-in-1 Phone System 1999] case studies are applied to evolve a peer-to-peer architecture to a client-server architecture. We have specified the architecture descriptions as component-connectors model of P2P-AP system as a graph. Graph-based modelling of architecture - already explained in Chapter 2 - allows us to exploit graph transformation for architecture evolution. Additional details about the component-connector view of P2P-AS architecture are provided in **Appendix B**. We asked the participants to use the PatEvol prototype for pattern-based architecture evolution with details in **Appendix E**

Evolution Scenarios, Change Primitives and Patterns

After presenting the evolution scenario in Figure 9.10, we now provide a mapping of the evolution scenario (evolution problem) and the necessary change primitives and change patterns (as available solutions) in Table 9.6. Please note, that problem solution mapping (evolution problem and available solution in terms of change pattern) is already discussed in Chapter 8.

Also, the technical distinction between change primitives and change patterns is already presented in Chapter 6. In Table 9.6, we only highlight the pattern as a reusable solution to recurring architectural problems.

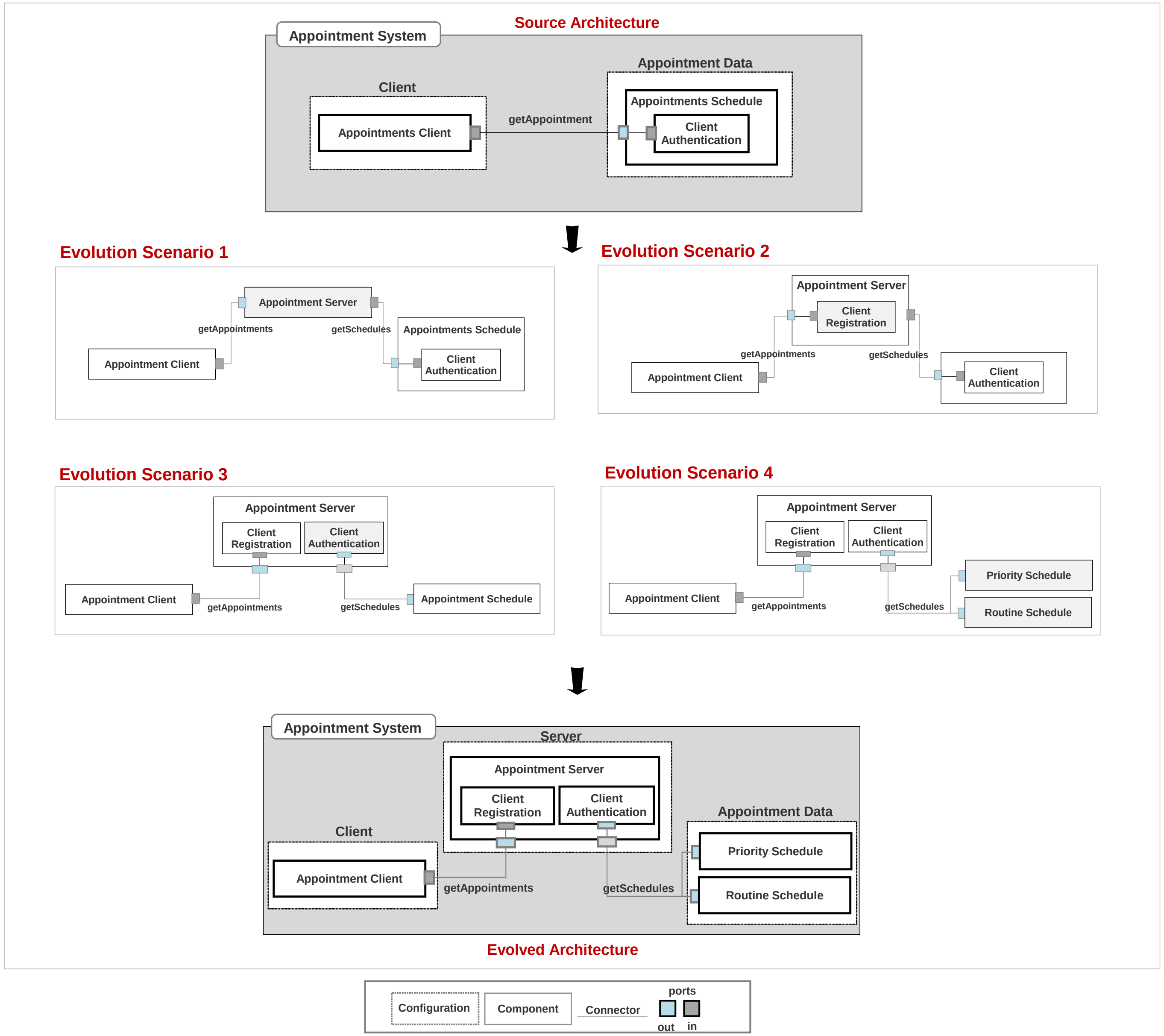


Figure 9.10: Source and Evolved Architecture Model with Architecture Evolution Scenarios.

<table>
<tr><th colspan="2">Evolution Scenario 1</th></tr>
<tr><td colspan="2">To interpose the AppointmentServer component between the AppointmentClients and AppointmentSchedule components. The newly integrated Appointment Server component mediates between the client requests and appointment scheduling.</td></tr>
<tr><th>Change Primitives</th><th>Change Pattern</th></tr>
<tr><td>CS-AS architecture is modified with addition of a new component AppointmentServer and two connectors (getAppointment, getSchedule)
to enable mediation among Clients and Appointment components.
$opr1 := ADD(AppointmentServer \in CMP)$
$opr2 := ADD(getAppointment((AppointmentClient, AppointmentServer) \in CMP) \in CON)$
$opr3 := ADD(getSchedule((AppointmentServer, AppointmentSchedule) \in CMP) \in CON)$
$opr4 := REM(getAppointment((AppointmentClient, AppointmentServer) \in CMP) \in CON)$</td><td>$ComponentMediation([C_M] < C_1, C_M, C_2 >)$
To interpose a mediator component (CM) among two or more directly connected components (C_1, C_2).</td></tr>
<tr><th colspan="2">Evolution Scenario 2</th></tr>
<tr><td colspan="2">To create a child component ClientRegistration inside the AppointmentServer component. The newly added Client Registration component enables registration of individual clients on the server.</td></tr>
<tr><th>Change Primitives</th><th>Change Pattern</th></tr>
<tr><td>CS-AS architecture is modified by creating the ClientRegistration component (atomic component) in Appointment Server (composite component) and a connector (register).
$opr1 := ADD(ClientRegister \in CMP)$
$opr2 := ADD(register((ClientRegister, AppointmentClient) \in CMP) \in CON)$</td><td>$ChildCreation([C] < X_1 : C >)$
To create a child component (X_1) inside an atomic component (C).</td></tr>
<tr><th colspan="2">Evolution Scenario 3</th></tr>
<tr><td colspan="2">To move a child component ClientAuthentication from AppointmentSchedule component to Appointment Server component. The addition of Client Authentication authentication of individual clients on the server before making an appointment.</td></tr>
<tr><th>Change Primitives</th><th>Change Pattern</th></tr>
<tr><td>CS-AS architecture is modified by moving ClientAuthentication component from AppointmentSchedule AppointmentSchedule to AppointmentServer . AppointmentSchedule is now an atomic component component and AppointmentServer is a composite component.
$opr1 := REM(ClientAuthentication \in CMP, AppointmentSchedule \in CMP)$
$opr2 := ADD(ClientAuthentication \in CMP, AppointmentServer \in CMP)$</td><td>$ChildAdoption([C] < X_1 : C >)$
To create a child component ($X1$) inside an atomic component (C).</td></tr>
<tr><th colspan="2">Evolution Scenario 4</th></tr>
<tr><td colspan="2">To replace an existing component AppointmentSchedule with two new components PrioritySchedule and RoutineSchedule. The newly added components provide either a priorotised or a routine scheduling based on the client request to the AppointmentServer.</td></tr>
<tr><th>Change Primitives</th><th>Change Pattern</th></tr>
<tr><td>CS-AS architecture is modified by replacing the AppointmentSchedule component with two newly added components PrioritySchedule and RoutineSchedule and connectors (getpriority, getRoutine).
$opr1 := ADD(RoutineSchedule \in CMP)$
$opr2 := ADD(PrioritySchedule \in CMP)$
$opr3 := ADD(getRoutine((RoutineSchedule, AppointmentServer) \in CMP) \in CON)$
$opr4 := ADD(getPriority((PrioritySchedule, AppointmentServer) \in CMP) \in CON)$
$opr5 := REM(getSchedule((AppointmentSchedule, AppointmentServer) \in CMP) \in CON)$
$opr6 := REM(AppointmentSchedule \in CMP) \in CON)$</td><td>$ActiveDisplacement(< C_1 : C_2 >, < C_1 : C_3 > [C_2 : C_3])$
To replace an existing component (C_1) with a new component (C_3) while maintaining the interconnection with with existing component (C_2).</td></tr>
</table>

Table 9.5: A Summary of Evolution Scenarios, Change Primitives and Change Patterns.

In Table 9.6, first we present the description of evolution scenario that follows the presentation of change primitives and finally the pattern as reusable solutions to address the evolution scenario.

Change Primitive

Represent a collection of composite change operations to enable addition, removal and modification of individual components and connectors. For example, in Evolution Scenario 1 (Table 9.6) change primitive requires at-least a total of 4 change operations to integrate a mediator component in existing architecture. We only consider changes on architectural components and connectors omitting changes on ports and endpoints - it has already been explained that components must contains ports and connectors must contain endpoints (detailed in Chapter 6).

9.7.2 Summary of Comparison for Primitive vs Pattern-based Changes

After presenting evolution scenarios and patterns to address these scenarios, we discuss the results of evaluation. A summary of the results of evaluation is presented in Table 9.6. In Table 9.6, we compare the efficiency of change implementation using change primitives and change patterns using:

1. **Total Change Operations -** to quantify the required efforts for change implementation, we count the number of change operators required for implementing a change and call this Total Change Operations (TCO). TCO is defined as *the total number of architecture change operations required to resolve an architecture evolution scenario.*

   For example, in Table 9.6 the TCO value for component integration is 4. The TCO concept is inspired by **Line of Code (LOC)** methods. The difference between the proposed TCO and LOC is that TCO is a measure of the total number of change operations to implement a particular change. LOC is a measure of the total lines to code required to achieve a functionality. LOC is a measure of the size of source code in terms of executable lines. In contrast, the TCO measures the operational complexity for architecture evolution.

2. **Total Time Taken -** represents the time efficiency of change implementation. Time efficiency is more relevant during dynamic adaptation or evolution of time critical software. Here we provide a comparison of the time efficiency of primitive and pattern-based changes. For example, in Table 9.6, time required to integrate a component using primitive change is 231 seconds. Alternatively, the application of component Mediation pattern can achieve the same effect in 42 seconds (only 25% time taken when compared to primitive changes).

3. **Ratio of Change Operationalisation (Primitive vs Pattern) -** represents the ratio of change operators from Pattern to primitive changes expressed as: $1 - (\frac{N_{TCO}}{E_{TCO}})$. For an example, see Table 9.6. The terms N_{TCO} denotes the number of change operations required by the patterns (N), whereas E_{TCO} denotes the number of change operations required by the primitive (E).

4. **Ratio of Time Taken (Primitive vs Pattern) -** represents the ratio of change operators from pattern to primitive changes expressed as: $1 - (\frac{N_{Time}}{E_{Time}})$. For example, in Table 9.6 as below. The terms N_{Time} denotes the total taken by the patterns (N) to implement a change, whereas E_{Time} denotes the total time taken by the primitive (E).

Change Pattern			**Change Primitive**			**Efficiency Comparison**	
Pattern Name	*TCO*	*Time Taken*	*Intent of Primitive*	*TCO*	*Time Taken*	$1 - (\frac{N_{TCO}}{E_{TCO}})$	$1 - (\frac{N_{Time}}{E_{Time}})$
Component Mediation	3	42	Integration of Components	4	156	25	27
Parallel Mediation	3	38	Integration of Components	3	105	0	36
Correlated Mediation	3	67	Integration of Components	8	440	63	15
Functional Slicing	3	33	Splitting of Components	4	101	25	32
Functional Unification	3	37	Merging of Components	4	96	25	38
Active Displacement	3	54	Replacement of Components	4	177	25	30
Child Creation	3	34	Composition of Components	4	94	25	36
Child Adoption	3	41	Move a Component	6	143	50	28
Child Swap	3	48	Swap a Component	4	149	25	32
	3	43.77		4.55	162.33	29.22%	30.44%

Table 9.6: A Summary of Efforts for Change Primitives and Change Patterns.

Based on the summary of results in Table 9.6 we provide an overview of the comparative analysis for TCO for primitive and pattern-based changes in Figure 9.11. In addition, we also provide a comparison of time taken during primitive as well as pattern-based changes in Figure 9.12. The graph in Figure 9.11 reflects that the operation using pattern-based changes is a constant. In contrast, primitive changes requires between 3 and 8 change operations. In addition, pattern-based changes provide a process-based overview of change implementation. As highlighted in Table 9.6, the pattern-based changes on average require 43.77 seconds (43.77/60 = 0.73 minutes), whereas the primitive changes in comparison require 162.33 seconds (162.33/60 = 2.70 minutes). The results suggests that:

Pattern-based changes take only 29% of change operations compared to primitive changes. However, pattern-based change does not support a fine granular change representation.

Based on the summary of results in Table 9.6 we provide an overview of analysis for time taken for primitive as well as pattern-based changes.

In pattern-based changes it takes on average less than a minute to resolve an evolution scenario. Primitive changes on average take more than 2.50 minutes to implement the change. Pattern-based changes on average require only 30% of time compared to primitive changes.

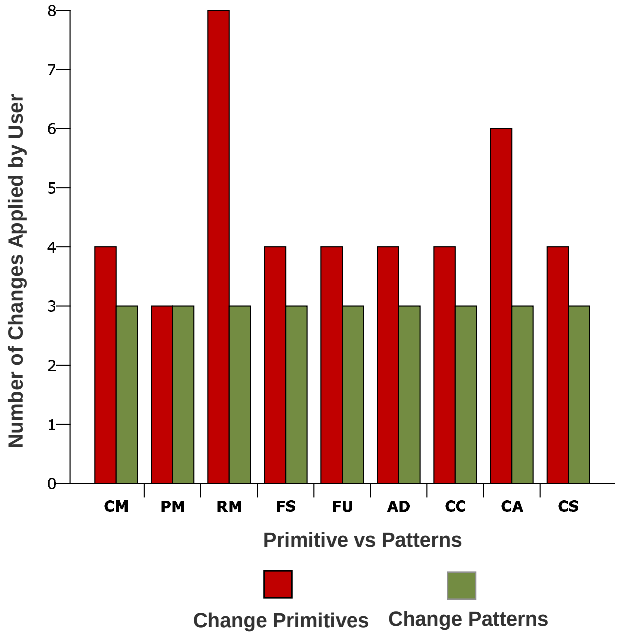


Figure 9.11: A Comparison of TCO for Pattern vs Primitive Changes.

9.7.3 Granularity vs Reusability of Changes

In primitive vs pattern-based changes, there is a trade-off between the granularity and reusability of architectural changes. Granularity of architectural changes refer to the completeness of changes (e.g. adding configurations with components that contain ports). Reusability of architectural changes refers to reuse of generic change operations (e.g. integration, composition of components). To discuss granularity vs reusability we represent the types of architectural changes as a layered structure with primitive changes (a.k.a. change operations) at the bottom that are abstracted by change patterns at the top.

For example, moving from top to bottom (patterns to primitives) the granularity of change is increased. The loss of granularity results in:

1. *Change Implementation at Higher Abstraction* - patterns with reusable but coarse-grained changes only provide generic changes that affect components and connectors. This abstrac-

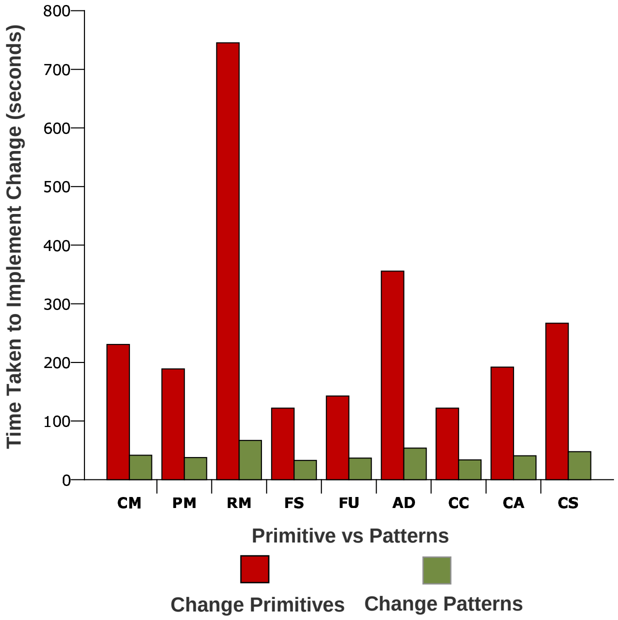


Figure 9.12: An Overview of Time Taken for Primitive vs Pattern-based Changes.

tion do not support lower level changes changes at the component operations level, that are exposed at ports. In contrast, the change primitives supported with change operations support a fine granular change representation. The granularity of change implementation is also a concern of source code level changes [Williams 2010, Buckley 2005] and not the architecture evolution.

2. *Structural Integrity of Architecture Model* - the granularity of architectural changes ensure that architectural integrity is preserved (components and their port, connectors have bindings). In our solution, architectural hierarchy is preserved with change operations that are abstracted in patterns.

In contrast to primitive architectural changes, pattern-based changes support reuse that results in an increased efficiency of the architecture evolution process - 30 % less effort for change implementation and 80% less time required to implement changes. However, pattern-based changes support reuse of architectural evolution but do not support a fine-granular change implementation.

9.8 Threats to Validity of Research

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.

Challenges of Software Architecture Evolution

Software architecture evolution involves different challenges that include modelling, analysing and executing architectural changes in a consistent and efficient manner and empowering the role of software architects to evolve architectures in a semi-automated way [Barnes 2013, Bennett 2000]. For example, in our case the ultimate benefits of our solution can only be practically quantified if utilising our solution produces better results than already existing solutions in an industrial context. However, in an industrial scale software architecture [Clerc 2007, Slyngstad 2008] evolution usually takes place over long periods that span months, years and often decades [Slyngstad 2008].

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. In the general context of this thesis, 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 reusablity and efficiency of the change execution process.

Threat I - Enabling the Continuity of Pattern Discovery Process

During the architecture change mining process, *continuity of pattern discovery* refers to providing the necessary methods and techniques to continuously discover architecture change patterns over time and from different change logs. In this thesis, a more rigorous validation of solution requires more case studies to discover patterns. Currently, in the PatEvol framework - architecture change mining process - we only have two case studies to investigate their evolution and to discover architecture change patterns.

To minimise this type of threat, there is a need to acquire more data from different logs and case studies. Data collection from representative sources is time-intensive process requiring months or years for the acquisition of representative data [Kagdi 2007, Buckley 2005]. To compensate for this, pattern discovery algorithms (cf. **Chapter 7**) can be seen as a solution to minimise this type of threat. Pattern discovery from change log graph is an automated and user customised technique to continuously discover patterns as new log data becomes available.

Threat II - Evaluating the Accuracy of Pattern Selection Process with New Patterns

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 language. This threat has a direct impact on selecting the most appropriate patterns from pattern language. Currently, we have a total of (7+2), i.e., 7 change patterns and 2 variants of patterns that represent a relatively limited number of patterns. As the number of change patterns in the pattern language grows it may have an impact on the precision of pattern selection.

We have classified patterns into three different categories that helps to increase the pattern selection precision (similar patterns are grouped together). If pattern discovery is supported as a continuous process, the number of patterns in the pattern language is expected to grow over time and that requires a re-evaluation of the accuracy of pattern selection in future. The presented QOC methodology in **Chapter 8** provides the user with an accurate and incremental means to select appropriate patterns.

Threat III - Limited Data Size and Practitioners' Experience

In the context of pattern mining and pattern application, we also need to consider the validity threats regarding i) *the size of the data* used for pattern mining and ii) *the experience of the practitioners* evaluating pattern application.

More specifically, the relative size of data in the change log used for pattern mining is smaller when compared to other solutions of pattern mining [Geng 2008] and repositories for source code analysis [Zimmermann 2005]. We have available data from two case studies of architectural evolution that represent a couple of thousands of changes (2200 approx. individual change operations). An inherent limitation with such small data size and its analysis lies with the discovery of a limited number of patterns. Moreover, any reliable cross validation of the mining techniques and solutions must rely on a significantly large data sets [Hassan 2008] - currently lacking in our evaluation. In the absence of a large data set, a possibility to minimse such threat is to use artificially generated data [Agrawal 1995] or use larger data set for cross validation. As part of future research (detailed in subsequent chapter), we aim to follows the later approach by means of customising and validating the pattern discovery algorithms by mining architectural change log of significant size [ROS-Distributions 2010].

Another threat relevant to user-based evaluation of the impact of patterns on architecture evolution is a limited experience of the practitioners' who participated in the evaluation. Specifically, due to time constraints and the willingness of the practitioners to participate in the evaluation we only had a total of five practitioners with a total combined inexperience of seven years in software architecture related activities. Such small size have direct implications on the user-based validation of pattern applicability. To compensate for a small population, we used the key informant method [Gallivan 2001] as a qualitative approach to ensure high-relevance of the participants to the evaluation. However, there is still a need to further evaluate the solution with more participants for a more objective interpretation of the results. A possibility lies with the deployment of a survey to engage geographically distributed participants to evaluate our approach.

Threat IV - The Adoption of Architectural Change Logs for Evolution Analysis

The research and practices on history-based analysis of software evolution are primarily focused on analysing changes in source code repositories [Robbes 2005] and architectural configurations [Van der Westhuizen 2002]. The notion of the change log for mining architectural evolution analysis is not well established with only a few studies exploiting the concept of architectural change log with release histories [Wermelinger 2011]. With a lack of evidence we face a threat to validity of research on mining change patterns from architectural change logs. More specifically, the question arises: *'what is the practicality and adoption of the architecture change logs for analysing architecture evolution histories'*?

The only answer to such a question lies with some recent research that exploited the concepts of architecture change logs for mining the evolution of Eclipse [Wermelinger 2011] and capturing the evolution of ROS [ROS 2010]. However, based on only a limited evidence the research cannot objectively argue about the adoption of change logs from a wider research community and by the practitioners as well. The concept of change log in this thesis complements the available evidence and outlines the necessary challenges and appropriate solutions for log-based mining of architectural evolution.

9.9 Chapter Summary

We utilised the ISO/IEC 9126 - 1 quality model to evaluate the efficiency and quality of the proposed solution. More specifically, in the PatEvol framework first we evaluate the architecture change mining process to address the challenges in RQ 1 and RQ 2 and then architecture change execution to address challenges in RQ 3 and RQ 4 (outlined in Chapter 1):

- **RQ 1** deals with evaluating the suitability and efficiency of modelling architectural changes as change log graphs discussed in Section 9.4. The evaluation suggests that as the size of data increases (more than **1200 changes**) in a log file, graph-based traversal and searching of log data is efficient in Section 9.4.

- **RQ 2** aims at evaluating the efficiency, accuracy of pattern discovery process. We have evaluated the discover algorithms in comparison to any manual process for pattern discovery and found that pattern discovery is more efficient and accurate than manual efforts discussed in Section 9.5.

- **RQ 3** is evaluated in terms of selecting the most appropriate change pattern from the pattern language. We observed that pattern classification helps in increasing the precision of pattern selection discussed in Section 9.6.

- **RQ 4** is evaluated based on evaluating the efficiency of pattern-based architecture evolution process. The evaluation suggests that pattern-based evolution is more efficient than primitive changes discussed in Section 9.7.

Chapter 10

Conclusions and Future Research

Contents

10.1 Research Focus and Implications of the PatEvol Framework

In this chapter, we present the main conclusions of our research by highlighting the contributions and discussing future research. First, we provide a summary of the research focus and practical implementation of the PatEvol framework. Then, we discuss the core contributions of this research followed by dimensions of future research.

In modern day software, we face a challenge with frequently evolving requirements that needs to be implemented in existing software in a timely and cost-effective manner [Garlan 2009, Tamzalit 2010]. Lehman's law of continuing change [Lehman 1996] poses a direct challenge for research and practices that aim to support long-living and continuously evolving architectures [Le Goaer 2008] under changing requirements [Yskout 2012]. The primary challenges (as identified in **Chapter 3**) to support a continuous change are concerned with: a) acquisition and application of reusable solutions to address recurring evolution problems and b) selection of an appropriate abstraction for software change implementation [Medvidovic 1999]. To address these challenges, we support the discovery of evolution-centric knowledge that can be reused to evolve software at its architecture level. Some industrial studies [Cámara 2013, Mohagheghi 2004] have suggested that *the integration of an empirically discovered reuse knowledge in the architecture evolution process supports reusability of change implementations and ultimately the efficiency of evolution process*. The focus of this research was to discover reuse knowledge and expertise - operationalisation and patterns - that can be integrated in the architecture evolution process (**Chapter 4**). In architecture change mining process, we performed a post-mortem analysis of architecture evolution histories (**Chapter 5**) - change logs - to discover recurring operationalisations (**Chapter 6**) and change patterns (**Chapter 7**). In the change execution process (**Chapter 8**), we applied the discovered operationalisation and patterns to support reuse in the evolution of software architectures.

In terms of the results, the research also draws inspiration from pattern languages to build complex architectures in the real world [Alexander 1999]. We focus on composition and the application of a pattern language that exploits a collection of discovered patterns and their relations to evolve software architectures. Language composition is enabled with a continuous discovery of patterns from architecture change logs and formalisation of relations among discovered change patterns. The language application is supported with an incremental selection and application of patterns to achieve reuse in architecture-centric software evolution. Reuse-knowledge in the proposed pattern language is expressed as a collection of connected patterns (a.k.a. pattern relations). The application domain of the pattern language is component-based software architectures and their evolution. Graph mining is exploited for pattern discovery [Agrawal 1995] (language composition) and graph transformation for pattern-driven architecture evolution [Bhattacharya 2012] (language application).

10.1.1 Practical Implementation of the PatEvol Framework

In recent years, the needs for reuse knowledge and expertise have grown as indicated in research [Garlan 2009, Tamzalit 2010] and practice [Cámara 2013] for software architecture evolution. This research provides a framework and its implementation with two software prototypes to (semi-) automate the *architecture change mining* and *architecture change execution* processes.

We have developed a prototype *GPride* (Graph-based Pattern Identification) to support automation of the pattern discovery process. The input to the prototype is a log file (modelled as a log graph) for pattern discovery. The prototype supports a modular solution to pattern discovery by offering the pattern discovery algorithms that support parametrisation and customisation of the pattern discovery process. The output of the prototype is a list of discovered patterns that are specified in a pattern template for later reuse. The prototype *GPride* discovers exact as well as in-exact pattern instances from logs where only central pattern features suffice for identification as detailed in Chapter 7.

We also provide a prototype *PatEvol* (Pattern-based Architecture Evolution) to support pattern-driven reuse in architecture-centric software evolution. The input to the prototype is a list of discovered patterns and descriptions of the source architecture model that needs to be evolved. The prototype allows the user to select the most appropriate patterns in a given evolution scenario. Finally, the source architecture is transformed towards a target or evolved architecture using change patterns - the outcome is an evolved architecture model as detailed in Chapter 8.

The prototypes emphasise the needs for practical solutions supporting reusability on architecture evolution process [Clerc 2007, Slyngstad 2008]. Currently, we have implemented these two prototypes as standalone applications. However, we plan to provide a unified solution by combining the GPride and PatEvol prototype as an Eclipse plug-in[1] to enhance prototype usability.

10.2 Summary of Research Contributions

This research contributes a pattern language as a collection of architecture change patterns to promote reuse in the evolution of component-based software architectures. Pattern discovery is enabled by analysing and mining recurring architectural changes from change logs. Architecture change patterns abstract the primitive changes (addition, removal, modification of components and connectors) into reusable pattern-based changes (composition, decomposition, replacement etc. of components and connectors). We highlight the contributions of this thesis as:

- **A Systematic Review of Research on Architecture Evolution Reuse Knowledge:** We provided a systematic review of existing research to identify and classify the available evidence about evolution reuse in software architectures, and provided a comparison of existing research to highlight its potential, limitations and future dimensions. Chapter 3 can be viewed as a stand-alone contribution as the literature base. It helps with knowledge sharing to ACSE researchers and practitioners [Stammel 2011] and presents a collective impact of existing research and insights into dimensions of future research. The results of this contribution are published in [Ahmad 2014d].

- **A Framework for Acquisition and Application of Evolution Reuse Knowledge:** We have

[1] Plug-ins Eclipse: http://www.eclipse.org/resources/?category=Plug-ins

proposed a framework PatEvol that aims to unify the concepts of a) software repository mining and b) software evolution to enable acquisition and application of architecture evolution reuse knowledge. In the proposed PatEvol framework, we present knowledge acquisition (architecture change mining) to enable post-mortem analysis of evolution histories to discover evolution-centric knowledge. Furthermore, we support reuse of discovered knowledge to enable knowledge application (architecture change execution) that enables evolution-off-the-shelf in software architectures as presented in Chapter 4. The results of this contribution are published in [Ahmad 2011, Ahmad 2012e, Ahmad 2013b].

- **A Taxonomical Classification of Architecture Change Operationalisation:** By investigating architecture change logs, we taxonomically classify architectural change operations as *atomic*, *composite* and *sequential* type changes. We distinguish between the *change primitives* and *change patterns*. In addition, we classify operational dependencies as commutative and dependent type change operations to analyse the extent to which architecture change operations can be parallelised in Chapter 6. The results of this contribution are published in [Ahmad 2012b].

- **Mining Architecture Change Pattern from Logs:** We provided algorithms to discover architecture change patterns from logs. Pattern discovery algorithms when executed on change logs provide an automated and continuous discovery of patterns. Scalability of pattern-discovery process beyond manual analysis is supported with a prototype 'G-Pride' (Graph-based Pattern Identification) enabling automation and parametrised user intervention for pattern mining in Chapter 7. The results of this contribution are published in [Ahmad 2012c, Ahmad 2013a].

- **Pattern-driven Reuse in Architecture Evolution:** The solution promotes architecture evolution as a two-step process: to leverage architectural change mining - discovering pattern instances from change logs - and to support potential reuse during architecture change execution. We demonstrated that *if an architectural evolution problem can be specified declaratively, then pattern-driven evolution could relieve an architect from the underlying operational concerns for executing routine evolution tasks facilitated with change patterns*. We provided a prototype 'Pat-Evol' (Pattern-based Architecture Evolution) that enables automation and user intervention for architecture change execution in Chapter 8. The results of this contribution are published in [Ahmad 2012a, Ahmad 2014b].

10.3 Dimensions of Future Research

We now discuss dimensions of possible future research that can complement the existing research on evolution reuse or outline challenges for some novel solutions.

10.3.1 Pattern-driven Plans for Architecture Evolution

An interesting aspect of future research is to possibly integrate the proposed change patterns in the evolution plans discussed in [Barnes 2013]. The architecture evolution plans [Barnes 2013] describe an approach for planning, modelling and reasoning about architecture evolution. Specifically, the evolution plan exploits the concept of evolution styles from [Garlan 2009] that empower the role of an architect to derive high-level, reusable paths of architecture evolution. These paths provide a decision support to the architect by evaluating different paths of evolution based on various trade-offs (cost vs time of evolution, etc.). One of the limitations of the evolution styles (the foundation for evolution plans) are derived based on the experience, observations and the expertise of the individual architects that limit their reusability across systems.

We believe, as part of the future research an interesting investigation lies with analysing the applicability of the empirically discovered patterns to derive evolution plans. This can ensure better re-usability both at evolution planning (with styles) and evolution execution (with patterns) levels. Moreover, the discovered patterns can also abstract the primitive and low-level architectural changes to go beyond deriving plans and also assist the architect with reusing change execution.

10.3.2 Post-mortem Analysis of Architecture Evolution Histories of Evolving Software

The applicability of the log-based pattern discovery algorithms beyond the existing solution needs to be evaluated on different systems that evolve continuously. This means we need to acquire an extensive real-world data that allows us to customise our proposed algorithms and to cross-validate the results of pattern mining - with potential future research detailed below.

Mining Architecture Change Logs for ROS

The applicability of the log-based pattern discovery algorithms beyond the existing solution needs to be evaluated on different systems that evolve continuously. This means we need to acquire an extensive real-world data that allows us to customise our proposed algorithms and to cross-validate the results of pattern mining. We are specifically interested in the postmortem analysis of the architectural evolution history (recorded in the change logs) of the Robot Operating Systems (ROS) [ROS 2010]. In particular, we view the change logs of ROS architecture as an ideal example - providing us publicly available data (from 2010 - to date) [ROS-Distributions 2010] with thousands of architectural changes to systematically analyse the evolution of ROS. Therefore, as part of the future research, we aim to discover and investigate:

- The evolution patterns and their impact on the structure of the ROS that helps us to view the most frequent structural changes of the architecture to predict futuristic evolution.
- The impact of evolution on the dependencies that exist among the architectural components of ROS. The dependency analysis allows us to study the co-evolution of architectural elements.
- The classical work on the laws of software evolution [Lehman 1996] and their implication can be empirically revisited in terms of a long-term evolving software.

Patterns for Legacy Modernisation towards Cloud-enabled Software

In recent years, there is a lot of attention on developing solutions that enable the migration of legacy systems towards cloud-enabled software. Specifically, cloud computing as a platform allows organizations to leverage the distributed and interoperable services to deploy their legacy (on-premise) software systems over publicly available resources. From a business point of view, organizations can benefit from the pay-per-use model offered by cloud services rather than an upfront purchase of costly and over-provisioned infrastructure. From a technical perspective, the scalability, interoperability, and efficient (de-)allocation of resources through cloud services can enable a smooth execution of organizational operations. However, legacy migration towards cloud requires an appropriate process and tool support. A recent review of research [Jamshidi 2013a] has highlighted the growing needs for reusable knowledge, processes and tool support for legacy cloudification. Moreover, the *Legacy-to-Cloud Migration Horseshoe* [Ahmad 2014a] as a conceptual framework provides a foundation for future integration of migration patterns for architecture-driven legacy migration.

As a possible dimension of future research, we primarily focus on pattern-driven reuse of architectural migration. Considering migration as a recurring problem, in future we aim to exploit the migration process patterns as reusable solutions to frequent problems of architectural migration. A migration process pattern is defined as a generic and repeatable solution that addresses the frequently occurring migration problems. Based on our proposed framework, we aim to investigate the architecture migration processes to empirically discover migration process patterns. A catalogue of migration process patterns is envisaged as a pattern collection that shall guide the migration process.

10.3.3 The Notion of Architecture Change Anti-Patterns

Central to a pattern-based design and evolution process is reusability and proven practices to effectively tackle recurring problems [Gamma 2001, Goedicke 2002]. However, the pitfalls or the negative consequences of applying patterns cannot be overlooked - resulting in the emergence of anti-patterns [Mowbray 1998]. An anti-pattern represents a frequent solution to a recurring problems but it has some negative consequences on software design. The consequences may result in violating the design constraints and compromise the philosophy of pattern-based design by producing negative impacts.

The role of the 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 [Szyperski 2002, Medvidovic 1999] models may be violated as a consequence of a pattern-based evolution. These *counter-productive and negative impacts of change patterns on architecture model results in change anti-patterns*. A detailed discussion of potential anti-patterns is beyond the scope of this research. However, we believe that in addition to discovering the patterns and their variants (positive impacts), a complementary future work on discovery of change anti-patterns (positive impacts) and possibly preventing them ensures the efficiency of evolution process and the structural integrity of evolved architecture.

Bibliography

[3-in-1 Phone System 1999] 3-in-1 Phone System. *K3: Cordless Telephony Profile*. Bluetooth Specification Version, vol. 1, 1999.

[Agrawal 1995] Rakesh Agrawal and Ramakrishnan Srikant. *Mining Sequential Patterns*. In In Eleventh International Conference on Data Engineering, (ICDE'95), pages 3–14. IEEE, 1995.

[Ahmad 2010] Aakash Ahmad and Claus Pahl. *Pattern-based Customisable Transformations for Style-based Service Architecture Evolution*. In In 6th International Conference on Next Generation Web Services Practices, pages 371–376. IEEE, 2010.

[Ahmad 2011] Aakash Ahmad and Claus Pahl. *Customisable Transformation-driven Evolution for Service Architectures*. In In 15th European Conference onSoftware Maintenance and Reengineering (CSMR), pages 373–376. IEEE, 2011.

[Ahmad 2012a] Aakash Ahmad, Pooyan Jamshid, Claus Pahl and Fawad Khaliq. *PatEvol - A Pattern Language for Evolution in Component-Based Software Architectures*. First Workshop on Patterns Promotion and Anti-patterns Prevention, 2012.

[Ahmad 2012b] Aakash Ahmad, Pooyan Jamshidi, Muteer Arshad and Claus Pahl. *Graph-based Implicit Knowledge Discovery from Architecture Change Logs*. In In Seventh Workshop on SHaring and Reusing Architecture Knowledge, pages 116–123. ACM, 2012.

[Ahmad 2012c] Aakash Ahmad, Pooyan Jamshidi and Claus Pahl. *Graph-based Pattern Identification from Architecture Change Logs*. In In Tenth International Workshop on System/Software Architecture, pages 200–213. Springer, 2012.

[Ahmad 2012d] Aakash Ahmad, Pooyan Jamshidi and Claus Pahl. *Reuse at Runtime: Towards a Pattern Language for Self-Adaptation in Software Architectures*. Technical Report: School of Computing, Dublin City University, 2012. Available from: `www.computing.dcu.ie/~pjamshidi/DAPL.pdf,(Accessed:27-08-2015)`.

[Ahmad 2012e] Aakash Ahmad and Claus Pahl. *Pat-Evol: Pattern-drive Reuse in Architecture-based Evolution for Service Software*. volume 88, pages 200–213. ERCIM News, 2012.

[Ahmad 2013a] Aakash Ahmad, Pooyan Jamshidi and Claus Pahl. *Graph-based Discovery of Architecture Change Patterns from Logs*. Technical Report: School of Computing, Dublin City University, 2013. Available from: `www.computing.dcu.ie/~pjamshidi/PatternDiscovery.pdf,(Accessed:22-06-2013)`.

[Ahmad 2013b] Pooyan Ahmad Aakash Jamshidi and Claus Pahl. *A Framework for Acquisition and Application of Software Architecture Evolution Knowledge*. In ACM SIGSOFT Software Engineering Notes, volume 38, 2013.

[Ahmad 2014a] Aakash Ahmad and Ali Babar. *A Framework for Architecture-driven Migration of Legacy Systems to Cloud-enabled Software*. In First Workshop on Software Architecture Erosion and Architectural Consistency. ACM, 2014.

[Ahmad 2014b] Aakash Ahmad, Pooyan Jamshid, Claus Pahl and Fawad Khaliq. *A Pattern Language for the Evolution of Component-based Software Architectures*. In Electronic Communications of the EASST, Special Issue on Patterns Promotion and Anti-patterns Prevention. ECEASST, 2014.

[Ahmad 2014c] Aakash Ahmad, Pooyan Jamshidi and Khaliq Fawad Pahl Claus. *A Pattern Language for the Evolution of Component-based Software Architectures*. Electronic Communications of the EASST, vol. 59, pages 1–31, 2014.

[Ahmad 2014d] Pooyan Ahmad Aakash Jamshidi and Claus Pahl. *Classification and Comparison of Architecture Evolution Reuse Knowledge - A Systematic Review*. In Journal of Software: Evolution and Process, volume 26, pages 654–691, 2014.

[Alexander 1979] Christopher Alexander. The Timeless Way of Building, volume 1. New York: Oxford University Press, 1979.

[Alexander 1999] Christopher Alexander. *The Origins of Pattern Pheory: The Future of the Theory, and the Generation of a Living World*. IEEE Software, vol. 16, no. 5, pages 71–82, 1999.

[Babar 2004] Muhammad Ali Babar, Liming Zhu and Ross Jeffery. *A Framework for Classifying and Comparing Software Architecture Evaluation Methods*. In In 2004 Australian Software Engineering Conference, pages 309–318. IEEE, 2004.

[Babar 2009] Muhammad Ali Babar. Software Architecture Knowledge Management. Springer, 2009.

[Baresi 2002] Luciano Baresi and Reiko Heckel. *Tutorial Introduction to Graph Transformation: A Software Engineering Perspective*. In Graph Transformation, pages 402–429. Springer, 2002.

[Baresi 2006a] Luciano Baresi, Elisabetta Di Nitto and Carlo Ghezzi. *Toward Open-world Software: Issue and Challenges*. IEEE Computer, vol. 39, no. 10, pages 36–43, 2006.

[Baresi 2006b] Luciano Baresi, Reiko Heckel, Sebastian Thöne*et al. Style-based Modeling and Refinement of Service-oriented Architectures*. Software & Systems Modeling, vol. 5, no. 2, pages 187–207, 2006.

[Barnes 2013] Jeffrey M Barnes and David 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, 2013.

[Barnes 2014] Jeffrey M Barnes, David Garlan and Bradley Schmerl. *Evolution Styles: Foundations and Models for Software Architecture Evolution*. volume 13, pages 649–678. Springer, 2014.

[Basili 1990] Victor R. Basili. *Viewing Maintenance as Reuse-oriented Software Development*. IEEE Software, vol. 7, no. 1, pages 19–25, 1990.

[Bengtsson 1999] P Bengtsson and Jan Bosch. *Architecture Level Prediction of Software Maintenance*. In In hird European Conference on Software Maintenance and Reengineering, pages 139–147. IEEE, 1999.

[Bengtsson 2004] PerOlof Bengtsson, Nico Lassing, Jan Bosch and Hans van Vliet. *Architecture-Level Modifiability Analysis (ALMA)*. Journal of Systems and Software, vol. 69, no. 1, pages 129–147, 2004.

[Bennett 2000] Keith H Bennett and Václav T Rajlich. *Software Maintenance and Evolution: A Roadmap*. In Conference on the Future of Software Engineering, pages 73–87. ACM, 2000.

[Bhattacharya 2012] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu and Michalis Faloutsos. *Graph-based Analysis and Prediction for Software Evolution*. In In 33rd International Conference on Software Engineering, pages 419–429. IEEE, 2012.

[Bjørnson 2008] Finn Olav Bjørnson and Torgeir Dingsøyr. *Knowledge Management in Software Engineering: A Systematic Review of Studied Concepts, Findings and Research Rethods Used*. Information and Software Technology, vol. 50, no. 11, pages 1055–1068, 2008.

[Bradbury 2004] Jeremy S Bradbury, James R Cordy, Juergen Dingel and Michel Wermelinger. *A Classification of Formal Specifications for Dynamic Software Architectures*. In International Workshop on Self-Managed Systems (WOSMS'04), 2004.

[Brandes Ulrick 2007] D. Wagner Brandes Ulrick M. Gaertler. *Experiments on Graph Clustering Algorithms*. In 11th Annual European Symposium on Algorithms. Lecture Notes in Computer Science, 2007.

[Brandes 2002a] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt and M Scott Marshall. *GraphML Progress Report - Structural Layer Proposal*. In Graph Drawing, pages 501–512. Springer, 2002.

[Brandes 2002b] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt and M Scott Marshall. *GraphML Progress Report - Structural Layer Proposal*. In Graph Drawing, pages 501–512. Springer, 2002.

[Breivold 2012] Hongyu Pei Breivold, Ivica Crnkovic and Magnus Larsson. *A Systematic Review of Software Architecture Evolution Research*. Information and Software Technology, vol. 54, no. 1, pages 16–40, 2012.

[Brereton 2007] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner and Mohamed Khalil. *Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain*. Journal of Systems and Software, vol. 80, no. 4, pages 571–583, 2007.

[Brinkkemper 1996] Sjaak Brinkkemper. *Method Engineering: Engineering of Information Systems Development Methods and Tools*. Information and Software Technology, vol. 38, no. 4, pages 275–280, 1996.

[Buckley 2005] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid and Günter Kniesel. *Towards a Taxonomy of Software Change*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 17, no. 5, pages 309–332, 2005.

[Buschmann 1999] Frank Buschmann. Pattern Oriented Software Architecture: A System of Patters. Ashish Raut, 1999.

[Buschmann 2007] Frank Buschmann, Kelvin Henney and Douglas Schimdt. Pattern Oriented Software Architecture, Vol. 5, volume 5. John Wiley & Sons, 2007.

[Cámara 2013] Javier Cámara, Pedro Correia, Rogério De Lemos, David Garlan, Pedro Gomes, Bradley Schmerl and Rafael 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, pages 13–22. IEEE, 2013.

[Carrière 1999] S Jeromy Carrière, Steven Woods and Rick Kazman. *Software Architectural Transformation*. In Sixth Working Conference on Reverse Engineering, pages 13–23. IEEE, 1999.

[Chapin 2001] Ned Chapin, Joanne E Hale, Khaled Md Khan, Juan F Ramil and Wui-Gee Tan. *Types of Software Evolution and Software Maintenance*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 13, no. 1, pages 3–30, 2001.

[Clements 2003] Paul Clements, David Garlan, Reed Little, Robert Nord and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. In 25th International Conference on Software Engineering, 2003, pages 740–741. IEEE, 2003.

[Clerc 2007] Viktor Clerc, Patricia Lago and Hans van Vliet. *The Architect's Mindset*. In Software Architectures, Components, and Applications, pages 231–249. Springer, 2007.

[Conte 2004] Donatello Conte, Pasquale Foggia, Carlo Sansone and Mario Vento. *Thirty Years of Graph Matching in Pattern Recognition*. International Journal of Pattern Recognition and Artificial Intelligence, vol. 18, no. 3, pages 265–298, 2004.

[Côté 2007] Isabelle Côté, Maritta Heisel and Ina Wentzlaff. *Pattern-based Evolution of Software Architectures*. In In First European Conference on Software Architecture, pages 29–43. Springer, 2007.

[Davison 2004] Robert Davison, Maris G Martinsons and Ned Kock. *Principles of Canonical Action Research*. Information Systems Journal, vol. 14, no. 1, pages 65–86, 2004.

[Desrosiers 2011] Christian Desrosiers, Philippe Galinier, Alain Hertz and Pierre Hansen. *Improving Constrained Pattern Mining with First-fail-based Heuristics*. Data Mining and Knowledge Discovery, vol. 23, no. 1, pages 63–90, 2011.

[Dong 2006] Jing Dong, Yongtao Sun and Yajing Zhao. *Design Pattern Detection by Template Matching*. In Proceedings of the 2008 ACM symposium on Applied Computing, pages 765–769. ACM, 2006.

[Ducasse 2009] Stéphane Ducasse and Damien Pollet. *Software Architecture Reconstruction: A Process-oriented Taxonomy*. IEEE Transactions on Software Engineering, vol. 35, no. 4, pages 573–591, 2009.

[EBPPCaseStudy] EBPPCaseStudy. *NACHA - The Electronic Bill Presentment and Payment*. Available from: www.nacha.org,(Accessed:13-08-2011).

[Ehrig 2004] Hartmut Ehrig, Ulrike Prange and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. Springer, 2004.

[Ehrig 2006] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange and Gabriele Taentzer. Fundamentals of Algebraic Graph Transformation, volume 373. Springer, 2006.

[Erl 2009a] Thomas Erl. SOA Design Patterns. Prentice Hall, 2009.

[Erl 2009b] Thomas Erl. SOA Design Patterns. Prentice Hall, 2009.

[Fahmy 2000] Hoda Fahmy and Richard C Holt. *Using Graph Rewriting to Specify Software Architectural Transformations*. In Fifteenth IEEE International Conference on Automated Software Engineering, 2000, pages 187–196. IEEE, 2000.

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

[Flyvbjerg 2006] Bent Flyvbjerg. *Five Misunderstandings About Case-study Research*. Qualitative inquiry, vol. 12, no. 2, pages 219–245, 2006.

[Gall 1997] Harald Gall, Mehdi Jazayeri, Rene R Klosch and Georg Trausmuth. *Software Evolution Observations Based on Product Release History*. In International Conference on Software Maintenance, ICSM'97, pages 160–166. IEEE, 1997.

[Gallivan 2001] Michael J Gallivan. *Organizational Adoption and Assimilation of Complex Technological Innovations: Development and Application of a New Framework*. vol. 32, no. 3, 2001.

[Gamma 2001] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Abstraction and Reuse of Object-oriented Design. Springer, 2001.

[Ganek 2003] Alan G Ganek and Thomas A Corbi. *The Dawning of the Autonomic Computing Era*. IBM Systems Journal, vol. 42, no. 1, pages 5–18, 2003.

[Garg 2008] Amit X Garg, Dan Hackam and Marcello Tonelli. *Systematic Review and Meta-analysis: When one Study is Just Not Enough*. Clinical Journal of the American Society of Nephrology, vol. 3, no. 1, pages 253–260, 2008.

[Garlan 2004] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl and Peter Steenkiste. *Rainbow: Architecture-based Self-adaptation With Reusable Infrastructure*. IEEE Computer, vol. 37, no. 10, pages 46–54, 2004.

[Garlan 2009] David Garlan, Jeffrey M Barnes, Bradley Schmerl and Orieta 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, pages 131–140. IEEE, 2009.

[Geng 2008] Xiangjun Dong He Jiang Geng Runian and Wenbo Xu. *WTSPMiner: Efficiently Mining Weighted Sequential Patterns from Directed Graph Traversals*. In Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues, pages 372–379. Springer Berlin Heidelberg, 2008.

[Ghazizadeh 2002] Shayan Ghazizadeh and SudarshanS. Chawathe. *SEuS: Structure Extraction Using Summaries*. In Discovery Science, volume 2534 of *Lecture Notes in Computer Science*, pages 71–85. Springer Berlin Heidelberg, 2002.

[Gîrba 2006] Tudor Gîrba and Stéphane Ducasse. *Modeling History to Analyze Software Evolution*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, no. 3, pages 207–236, 2006.

[Goedicke 2002] Michael Goedicke and Uwe Zdun. *Piecemeal Legacy Migrating With an Architectural Pattern Language: A Case Study*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 14, no. 1, pages 1–30, 2002.

[Gomaa 2010] Hassan Gomaa, Koji Hashimoto, Minseong Kim, Sam Malek and Daniel A Menascé. *Software Adaptation Patterns for Service-oriented Architectures*. In 2010 ACM Symposium on Applied Computing, pages 462–469. ACM, 2010.

[Graaf 2007] Bas Graaf. *Model-driven Evolution of Software Architectures*. In 11th European Conference on Software Maintenance and Reengineering, CSMR'07, pages 357–360. IEEE, 2007.

[H. Tong 2007] B. Gallagher H. Tong C. Faloutsos and T. Eliassi-Rad. *Fast Best-Effort Pattern Matching in Large Attributed Graphs*. In 13th ACM International Conference on Knowledge Discovery and Data Mining, 2007.

[Harrison 2007] Neil B Harrison, Paris Avgeriou and Uwe Zdlin. *Using Patterns to Capture Architectural Decisions*. IEEE Software, vol. 24, no. 4, pages 38–45, 2007.

[Hassan 2008] Ahmed. E. Hassan. *The Road Ahead for Mining Software Repositories*. In Frontiers of Software Maintenance, pages 48–57. IEEE, 2008.

[Hentrich 2006] Carsten Hentrich and Uwe Zdun. *Patterns for Process-Oriented Integration in Service-Oriented Architectures.* In EuroPLoP, pages 141–198, 2006.

[Huang 2003] Yin-Fu Huang and Shao-Yuan Lin. *Mining Sequential Patterns using Graph Search Techniques*. In 27th Annual International Computer Software and Applications Conference, pages 4–9, 2003.

[Hudlicka 1996] Eva Hudlicka. *Requirements Elicitation with Indirect Knowledge Elicitation Techniques: Comparison of Three Methods*. In Second International Conference on Requirements Engineering, pages 4–11. IEEE, 1996.

[ISO-IEC-IEEE42010 2011] ISO-IEC-IEEE42010. *Systems and Software Engineering–Architecture Description*. Technical report, ISO/IEC/IEEE 42010, 2011.

[ISO/IEC25010 2010] ISO/IEC25010. *25010 Systems and Software Engineering–Systems and Software Quality Requirements and Evaluation (SQuaRE)–System and Software Quality Models*, 2010.

[Jamshidi 2013a] Pooyan Jamshidi, Ahmad Aakash and Claus Pahl. *Cloud Migration Research: A Systematic Review*. IEEE Transactions on Cloud Computing, vol. 1, no. 1, 2013.

[Jamshidi 2013b] Pooyan Jamshidi, Mohammad Ghafari, Ahmad Aakash and Claus Pahl. *A Framework for Classifying and Comparing Architecture-centric Software Evolution Research*. 2013.

[Javed 2009] Muhammad Javed, Yalemisew M Abgaz and Claus Pahl. *A Pattern-based Framework of Change Operators for Ontology Evolution*. In In On the Move to Meaningful Internet Systems: OTM 2009 Workshops, pages 544–553. Springer, 2009.

[Javed 2013] Muhammad Javed, Yalemisew M Abgaz and Claus Pahl. *Ontology Change Management and Identification of Change Patterns*. Journal on Data Semantics, pages 1–25, 2013.

[Jiang 2012] Chuntao Jiang, Frans Coenen and Michele Zito. *A Survey of Frequent Subgraph Mining Algorithms*. The Knowledge Engineering Review, vol. 1, no. 1, pages 1–31, 2012.

[Jung 2004] Ho-Won Jung, Seung-Gweon Kim and Chang-Shin Chung. *Measuring Software Product Quality: A Survey of ISO/IEC 9126*. IEEE Software, vol. 21, no. 5, pages 88–92, 2004.

[Kagdi 2007] Huzefa Kagdi, Michael L Collard and Jonathan I Maletic. *A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution*. Journal of Software Maintenance and Evolution: Research and Practice, vol. 19, no. 2, pages 77–131, 2007.

[Kampffmeyer 2007] Holger Kampffmeyer and Steffen Zschaler. *Finding the Pattern You Need: The Design Pattern Intent Ontology*. In Model Driven Engineering Languages and Systems, pages 211–225. Springer, 2007.

[Kandé 2000] Mohamed Mancona Kandé and Alfred Strohmeier. *Towards a UML Profile for Software Architecture Descriptions*. In UML 2000 - The Unified Modeling Language, pages 513–527. Springer, 2000.

[Kazman 1994] Rick Kazman, Len Bass, Mike Webb and Gregory Abowd. *SAAM: A Method for Analyzing the Properties of Software Architectures*. In Proceedings of the 16th International Conference on Software Engineering, pages 81–90. IEEE Computer Society Press, 1994.

[Kazman 1998] Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson and Jeromy Carriere. *The Architecture Tradeoff Analysis Method*. In 4th IEEE International Conference on Engineering of Complex Computer Systems, pages 68–78. IEEE, 1998.

[Kemerer 1999] Chris F. Kemerer and Sandra Slaughter. *An Empirical Approach to Studying Software Evolution*. IEEE Software, vol. 25, no. 4, pages 493–509, 1999.

[Ketkar 2005] Nikhil S. Ketkar, Lawrence B. Holder and Diane J. Cook. *Subdue: Compression-based Frequent Pattern Discovery in Graph Data*. In Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, pages 71–76, New York, NY, USA, 2005. ACM.

[Kiwelekar 2010] Arvind W Kiwelekar and Rushikesh K Joshi. *Ontological Analysis for Generating Baseline Architectural Descriptions*. In 4th European Conference on Software Architecture, pages 417–424. Springer, 2010.

[Kruchten 2006] Philippe Kruchten, Henk Obbink and Judith Stafford. *The Past, Present, and Future for Software Architecture*. IEEE Software, vol. 23, no. 2, pages 22–30, 2006.

[Kum 2006] Hye-Chung Kum, Joong Hyuk Chang and Wei Wang. *Sequential Pattern Mining in Multi-Databases via Multiple Alignment*. Data Mining and Knowledge Discovery, vol. 12, no. 2–3, pages 151–180, 2006.

[Lassing 2003] Nico Lassing, Daan Rijsenbrij and Hans van Vliet. *How Well can We Predict Changes at Architecture Design Time?* Journal of Systems and Software, vol. 65, no. 2, pages 141–153, 2003.

[Le Goaer 2008] Olivier Le Goaer, Dalila Tamzalit, Mourad Oussalah and A-D Seriai. *Evolution Shelf: Reusing Evolution Expertise Within Component-based Software Architectures*. In 32nd Annual IEEE International Computer Software and Applications, 2008. COMPSAC'08, pages 311–318. IEEE, 2008.

[Lehman 1996] Manny M Lehman. *Laws of Software Evolution Revisited*. In Software Process Technology, pages 108–124. Springer, 1996.

[Lehman 2003] Meir M Lehman and Juan F Ramil. *Software Evolution: Background, Theory, Practice*. Information Processing Letters, vol. 88, no. 1, pages 33–44, 2003.

[Lehnert 2012] Steffen Lehnert, Qurat Farooq and Matthias Riebisch. *A Taxonomy of Change Types and its Application in Software Evolution*. In 19th International Conference and Workshops on Engineering of Computer Based Systems, pages 98–107. IEEE, 2012.

[Leung 2005] Ho-pong Leung, Fu-lai Chung and Stephen Chi-fai Chan. *On the Use of Hierarchical Information in Sequential Mining-based XML Document Similarity Computation*. Knowledge and Information Systems, vol. 7, no. 4, pages 476–498, 2005.

[Li 2012] Zengyang Li, Peng Liang and Paris Avgeriou. *Application of Knowledge-based Approaches in Software Architecture: A Systematic Mapping Study*. Information and Software Technology, 2012.

[Loewe 1997] M Loewe, A Corradini, U Montanari, F Rossi, H Ehrig, R Heckel*et al.* *Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach*. 1997.

[Lytra 2012] Ioanna Lytra, Stefan Sobernig, Huy Tran and Uwe Zdun. *A Pattern Language for Service-Based Platform Integration and Adaptation*. 2012.

[MacLean 1991] Allan MacLean, Richard M Young, Victoria ME Bellotti and Thomas P Moran. *Questions, Options, and Criteria: Elements of Design Space Analysis*. Human Computer Interaction, vol. 6, no. 3-4, pages 201–250, 1991.

[MacLean 1995] Allan MacLean and Diane McKerlie. *Design Space Analysis and Use-representations*. Scenario-based Design: Envisioning Work and Technology in System Development, pages 183–207, 1995.

[Malavolta 2013] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione and Antony. Tang. *What Industry Needs from Architectural Languages: A Survey*. IEEE Transactions on Software Engineering, vol. 39, no. 6, pages 869–891, 2013.

[Medvidovic 1997] Nenad Medvidovic and Richard N Taylor. *A Framework for Classifying and Comparing Architecture Description Languages*. In IEEE Transactions on Software Engineering, volume 22, pages 60–76. Springer-Verlag, 1997.

[Medvidovic 1999] Nenad Medvidovic, David S Rosenblum and Richard N Taylor. *A Language and Environment for Architecture-based Software Development and Evolution*. In 1999 International Conference on Software Engineering, pages 44–53. IEEE, 1999.

[Medvidovic 2000] Nenad Medvidovic and Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70–93, 2000.

[Mens 1999] Kim Mens, Tom Mens, Bart Wouters and Roel Wuyts. *Managing Unanticipated Evolution of Software Architectures*. Lecture Notes in Computer Science, pages 75–75, 1999.

[Mens 2008] Tom Mens and Serge Demeyer. Software Evolution. Springer, 1st édition, 2008.

[Moghadam 2012] I Hemati Moghadam and Mel O Cinneide. *Automated Refactoring Using Design Differencing*. In 16th European Conference on Software Maintenance and Reengineering (CSMR), pages 43–52. IEEE, 2012.

[Mohagheghi 2004] Parastoo Mohagheghi and Reidar Conradi. *An Empirical Study of Software Change: Origin, Acceptance rate, and Functionality vs. Quality Attributes*. In International Symposium on Empirical Software Engineering, ISESE'04, pages 7–16. IEEE, 2004.

[Mooney 2013] Carl H. Mooney and John F. Roddick. *Sequential Pattern Mining – Approaches and Algorithms*. ACM Computing Surveys, vol. 45, no. 2, pages 1–39, 2013.

[Mowbray 1998] W. J. Brown Mowbray Thomas J. and H. W. McCornnick III. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley Sons, 1st édition, 1998.

[Nagappan 2013] Meiyappan Nagappan, Thomas Zimmermann and Christian Bird. *Diversity in Software Engineering Research*. In 9th Joint Meeting on Foundations of Software Engineering, pages 466–476. ACM, 2013.

[Pahl 2009] Claus Pahl, Simon Giesecke and Wilhelm Hasselbring. *Ontology-based Modelling of Architectural Styles*. Information and Software Technology, vol. 51, no. 12, pages 1739–1749, 2009.

[Pei 2002] Jian Pei, Jiawei Han and Wei Wang. *Mining Sequential Patterns with Constraints in Large Databases*. In Proceedings of the Eleventh International Conference on Information and Knowledge Management, pages 18–25. ACM, 2002.

[Perry 1992] Dewayne E Perry and Alexander L Wolf. *Foundations for the Study of Software Architecture*. ACM SIGSOFT Software Engineering Notes, vol. 17, no. 4, pages 40–52, 1992.

[Petticrew 2008] Mark Petticrew and Helen Roberts. Systematic Reviews in the Social Sciences: A Practical Guide. Wiley-Blackwell, 2008.

[Porter 2005] Ronald Porter, James O Coplien and Tiffany Winn. *Sequences as a Basis for Pattern Language Composition*. Science of Computer Programming, vol. 56, no. 1, pages 231–249, 2005.

[Robbes 2005] Romain Robbes and Michele Lanza. *Versioning Systems for Evolution Research*. In Proceedings of the Eighth International Workshop on Principles of Software Evolution, pages 155–164. IEEE Computer Society, 2005.

[ROS-Distributions 2010] ROS-Distributions. *ROS (Robot Operating System) Distributions*. 2010. Available from: http://wiki.ros.org/Distributions,(Accessed:15-06-2015).

[ROS 2010] ROS. *ROS (Robot Operating System) Distributions*. 2010. Available from: http://www.ros.org/,(Accessed:11-06-2015).

[Rosa 2004] Nelson Souto Rosa, Paulo Roberto Freire Cunha and George Roger Ribeiro Justo. *An Approach for Reasoning and Refining Non-functional Requirements*. Journal of the Brazilian Computer Society, vol. 10, no. 1, pages 59–81, 2004.

[Sadou 2005] Nassima Sadou, Dalila Tamzalit and Mourad Oussalah. *How to Manage Uniformly Software Architecture at Different Abstraction Levels*. In 24th International Conference on Conceptual Modeling, pages 16–30. Springer, 2005.

[Shaw 2006] Mary Shaw and Paul Clements. *The Golden Age of Software Architecture*. IEEE Software, vol. 23, no. 2, pages 31–39, 2006.

[Slyngstad 2008] Odd Petter N Slyngstad, Reidar Conradi, M Ali Babar, Viktor Clerc and Hans van Vliet. *Risks and Risk Management in Software Architecture Evolution: An Industrial Survey*. In 15th Asia-Pacific Software Engineering Conference, APSEC'08., pages 101–108. IEEE, 2008.

[Stammel 2011] Johannes Stammel, Zoya Durdik, Klaus Krogmann, Roland Weiss and Heiko Koziolek. Software Evolution for Industrial Automation Systems: Literature Overview. KIT, Fakultät für Informatik, 2011.

[Stojanović 2005] Zoran Stojanović and Ajantha Dahanayake. Service-oriented Software System Engineering: Challenges and Practices. Idea Group Publishing, 2005.

[Sun 2010] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen and Sai Zhang. *Change Impact Analysis Based on a Taxonomy of Change Types*. In 34th Annual IEEE Computer Software and Applications Conference (COMPSAC), pages 373–382. IEEE, 2010.

[Szyperski 2002] Clemens Szyperski, Dominik Gruntz and Stephan Murer. Component Software: Beyond Object-oriented Programming. Addison-Wesley, 2002.

[Taentzer 2000] Gabriele Taentzer. *AGG: A Tool environment For Algebraic Graph Transformation*. In Applications of Graph Transformations with Industrial Relevance, pages 481–488. Springer, 2000.

[Tamzalit 2010] Dalila Tamzalit and Tom Mens. *Guiding Architectural Restructuring Through Architectural Styles*. In 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), pages 69–78. IEEE, 2010.

[Tekumalla 2012] Bharath Tekumalla. *Master of Science Thesis in Software Engineering and Management, University of Gothenburg*. In 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS), pages 1–51, 2012.

[Tu 2002] Qiang Tu and Michael W Godfrey. *An Integrated Approach for Studying Architectural Evolution*. In 10th International Workshop on Program Comprehension, pages 127–136. IEEE, 2002.

[Ulrich 2010] William M Ulrich and Philip Newcomb. Information Systems Transformation: Architecture Driven Modernization Case studies. Morgan Kaufmann, 2010.

[van der Aalst 2002] Willibrordus Martinus Pancratius van der Aalst, Kees Max van Hee and Robert Arie van der Toorn. *Component-based Software Architectures: A Framework based on Inheritance of Behavior*. Science of Computer Programming, vol. 42, no. 2, pages 129–171, 2002.

[van der Hoek 2001] André van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel and Nenad Medvidovic. *Taming Architectural Evolution*. In 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, volume 26, pages 1–10. ACM, 2001.

[Van der Westhuizen 2002] Christopher Van der Westhuizen and André Van Der Hoek. *Understanding and Propagating Architectural Changes*. In Proceedings of the IFIP 17th World Computer Congress-TC2 Stream ystem Design, Development and Maintenance3rd IEEE/IFIP Conference on Software Architecture, pages 95–109, 2002.

[Wei 2007] Ong Kein Wei and Tang Mei Ying. *Knowledge Management Approach in Mobile Software System Testing*. In IEEE International Conference on Industrial Engineering and Engineering Management, pages 2120–2123. IEEE, 2007.

[Wermelinger 2011] Michel Wermelinger, Yijun Yu, Angela Lozano and Andrea Capiluppi. *Assessing Architectural Evolution: a case study*. volume 16, pages 623–666. Springer, 2011.

[Williams 2010] Byron J Williams and Jeffrey C Carver. *Characterizing Software Architecture Changes: A Systematic Review*. Information and Software Technology, vol. 52, no. 1, pages 31–51, 2010.

[Yskout 2012] Koen Yskout, Riccardo Scandariato and Wouter Joosen. *Change patterns: Co-evolving Requirements and Architecture*. Journal of Software and Systems Modeling, 2012.

[Yu 2009] Liguo Yu. *Mining Change Logs and Release Notes to Understand Software Maintenance and Evolution*. CLEI Electron Journal, vol. 12, no. 2, pages 1–10, 2009.

[Zdun 2007] Uwe Zdun. *Systematic Pattern Selection using Pattern Language Grammars and Design Space Analysis*. Software: Practice and Experience, vol. 37, no. 9, pages 983–1016, 2007.

[Zhang 2012] He Zhang and Muhammad Ali Babar. *Systematic Reviews in Software Engineering: An Empirical Investigation*. Information and Software Technology, 2012.

[Zimmermann 2003] Thomas Zimmermann, Stephan Diehl and Andreas Zeller. *How History Justifies System Architecture (or not)*. In Sixth International Workshop on Principles of Software Evolution, pages 73–83. IEEE, 2003.

[Zimmermann 2005] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber and Stephan Diehl. *Mining Version Histories to Guide Software Changes*. IEEE Transactions on Software Engineering, vol. 31, no. 6, pages 429–445, 2005.

Appendix A

Protocol and Auxiliary Information for Systematic Review

Contents

A.1 Scope of Systematic Literature Review

The primary objectives and general scope of this review is further clarified by establishing the PICOC (*Population, Intervention, Comparison, Outcome* and *Context)* perspectives [Petticrew 2008], summarised in Table E.3.1.

PICOC	SRQ-1	SRQ-2	SRQ-3
Population	Classification and Expression of Reuse in ACSE	Methods to enable reusable evolution and adaptation	Empirical approaches for reuse methods and techniques
Intervention	Taxonomical Classification of reuse methodologies	Identification of solutions for reuse-driven evolution and adaptation	Identification of reuse methods and techniques
Comparison	A holistic comparison among the population to analyse collective impact of existing research on methods and solution, formalism and tool support, research validations etc.		
Outcome	**Classification** and **comparison** framework with synthesised evidence to guide research and practices - Application of reuse-driven knowledge expertise in ACSE. - Empirical discovery of reuse-knowledge that can be shared and reused to guide ACSE.		
Context	A refined extension in our previous SLR [Williams 2010, Breivold 2012, Jamshidi 2013b] with an exclusive focus on evidence for reuse-driven evolution in architectures		

Table A.1: PICOC Criteria to define Scope and Goals of SLR.

A.2 Definition and Evaluation of the Review Protocol

According to the guidelines in [Brereton 2007], the review protocol drives the planning, conducting and documenting phases of the systematic review. The protocol[1] definition is provided in the reminder of this section. More specifically, we present i) identification of the needs and objectives for SLR, ii) definition of search strategies to identify, include and exclude and qualitatively analyse the relevant literature, iii) data extraction and results synthesis, and iv) results classification.

It aims to classify and compare existing research, identify the research potential, its limitations and outline future dimensions for methods, techniques and solution that enable evolution reuse in software architectures. In addition, the research questions help us to a) outline the scope and contributions of SLR and b) defining and evaluating the review protocol to conduct the SLR.

A.3 Conducting the Review

To conduct the review, we follow a three step process as i) searching the studies for review, ii) selection and qualitative assessment of studies, and iii) extraction and synthesis of data from studies.

The search terms used to identify primary studies were developed using suggestions in [Zhang 2012] and guided by the research questions. Our search process comprises of primary and secondary search.

- Primary Search is a five step process to identify and retrieve the relevant literature.

- Secondary Search includes a) review of references/bibliography section in the selected pri-

[1] We would like to acknowledge the efforts of Dr. Jim Buckely (affiliated with: Lero - the Irish Software Engineering Research Centre, University of Limerick, Ireland) and Bardia Mohabbati (affiliated with: Simon Fraser University, Canada) for their feedback and thoughtful suggestions throughout the development and evaluation of the review protocol.

mary studies to find other relevant articles, b) review of citations to the selected primary studies to find any relevant articles and c) identify and contact authors of selected primary studies for extended versions of the research, if required. The secondary search did not lead to identification of any relevant studies. The secondary search and study selection was performed iteratively until no new studies were found.

A.4 Literature Search Strategies

The search terms used were developed using suggestions in [Zhang 2012] and guided by the research questions. Our search process comprises of primary and secondary search.

Search Step	Description
Step 1 - Derive Search Strings	From SR-Qs and PICOC Criteria (cf. Table E.3.1)
Step 2 - **Consider Synonyms and Alternatives**	Consider alternative spellings and synonyms while composing search strings: - **Evolution** as [*change, restructure, update, extension, adaptation, reconfiguration, migration, transformation, modification*] - **Methods and Techniques to enable Reuse** as [*customise, pattern, plan, styles, framework, strategies*]. - **Empirical Methods for Discovery** as [*identification, extraction, tracing, mining, discovery, acquisition*] **Architecture** or **Software Architecture** [we only consider the term software architecture as only using architecture resulted in a large amount of irrelevant studies on Hardware, Network or System Architecture etc.]
Step 3 - **Search-term Combinations**	Boolean OR to incorporate alternative spellings and synonyms Boolean AND to link the major terms. Number of unique search string depends on a multiplier: ([AND] clause) x (<OR>-keywords)
Step 4 - **Search String Division**	Dividing strings so that they could be applied to different databases. Assigning unique to every (sub-) search string and customising them for all selected resources.
Step 5 - **Reference Management**	Citations with Zotero.

Table A.2: A Summary of the Step in Literature Search.

A.4.1 Executing Literature Search

The research question resulted in a composition of search string applied to 6 databases as illustrated in Figure A.1. We extracted the published peer-reviewed literature from years 1999 to 2012 (inclusive). The year 1999 was chosen as the preliminary search found no earlier results related to any of the research questions with 1550 manuscripts extracted. Because we used our primary search criteria on title and abstract, the results provided a relatively high number of irrelevant studies, which were further refined with secondary search. Note that we have decomposed the search string for illustrative reasons in Figure A.1. To search the primary studies the sub-strings in Figure A.1 were combined and represented as a single search string.

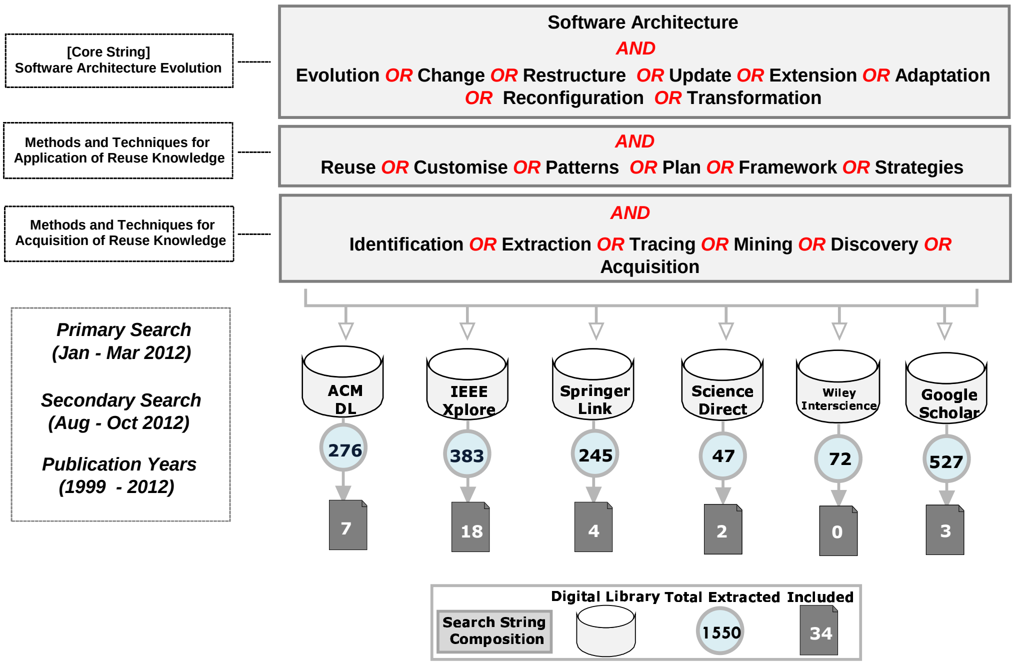

[Core String]
Software Architecture Evolution
Methods and Techniques for Application of Reuse Knowledge
Methods and Techniques for Acquisition of Reuse Knowledge
Software Architecture
AND
Evolution OR Change OR Restructure OR Update OR Extension OR Adaptation OR Reconfiguration OR Transformation
AND
Reuse OR Customise OR Patterns OR Plan OR Framework OR Strategies
AND
Identification OR Extraction OR Tracing OR Mining OR Discovery OR Acquisition
Primary Search (Jan - Mar 2012)
Secondary Search (Aug - Oct 2012)
Publication Years (1999 - 2012)
ACM DL
276
7
IEEE Xplore
383
18
Springer Link
245
4
Science Direct
47
2
Wiley Interscience
72
0
Google Scholar
527
3
Digital Library
Total Extracted
Included
Search String Composition
1550
34

Figure A.1: A Summary of the Primary Search Process.

A.5 Inclusion or Exclusion of Studies

The study selection phase comprised of two processes, initial selection and final selection in Table A.3, for the qualitative assessment.

1. *Initial Selection*: This process comprises screening of titles and abstracts of the potential primary studies. It was performed by the researchers against the inclusion or exclusion criteria in Table A.3. For almost 35% of studies, no decision could be made just on title and abstract, as these papers did not make a clear distinction between an explicit knowledge representation and application (SR-Q1 and SR-Q2) or acquisition (SR-Q1 and SR-Q3). In such cases, exclusion [NO] or proceeding to final selection [YES] involved examining the full text.

2. *Final Selection* This process is based on a brief validation scan of the studies, the use of formalisms and tool support and details of the experimental setup.

	Step 1 - Initial Selection	
SR-Q1 SR-Q2	Clear presentation of solution for application of evolution reuse-knowledge?	Clear presentation for discovery of evolution reuse-knowledge?
	(*If Yes, Goto Step 2: Otherwise Exclude Study*)	
	Step 1 - Final Selection	
SR-Q1 SR-Q2	1. Are findings in the study properly evaluated? 2. Formalism and tool support provided?	1. Source of knowledge and empirical discovery? 2. Details about experiment setup provided?
	(*If Yes, Include Study: Otherwise Exclude Study*)	

Table A.3: Summary of the 2 Step Study Selection Process.

A.6 Qualitative Assessment of Included Studies

For the 34 included studies, we primarily focused on the technical rigour of content presented in the study. We based our qualitative assessment on factors as *General Assessment* (G) and *Specific Assessment* (S), as summarised in Table A.4.

General Items for Quality Assessment (G)	
Score for General Items	[Yes = 1.0 Partially = 0.5 No = 0]
G1	Are problem definition and motivation of the study clearly presented?
G2	Is the research environment in which the study was carried out properly explained?
G3	Are research methodology and its organisation clearly stated?
G4	Are the contributions of the in-line with presented results?
G5	Are the insights and lessons learnt from the study explicitly mentioned?
S1	Is the research clearly focused on application or acquisition of evolution reuse?
S2	Are the details about related research clearly addressing evolution reuse in architectures?
S3	Is the research validation clearly illustrates application or acquisition of evolution reuse?
S4	Are the results clearly validated in a real (industrial case study) evaluation context?
S5	Are limitations and future implications for architecture evolution reuse clearly positioned?

Table A.4: Summary of Quality Assessment Checklist.

The quality assessment check-list is provided in Table A.4 and the quality ranking formula is as given as follows. G represents 5 factors as general assessment criteria from Table A.4, providing a maximum score of 1 (25% weight), S represents a total of 5 factors as specific items providing a maximum score of 3. S is weighted as 3 times more than G (75% weight) as specific contributions of a study are more important than general factors for assessment. Based on a consensus among the researchers and suggestions from the external reviewers, the criteria for qualitative assessment maximum score was $G + S = 4$ where a $3 - 4$ score represented quality papers, a score less than 3 and greater than or equal to 1.5 is acceptable and a score less than 1.5 results in exclusion.

$$QualityScore = \left[\frac{\sum_{G=1}^{5}}{5} + \frac{\sum_{S=1}^{5}}{5} \times 3 \right]$$

Quality ranking is an internal metric only that helps us to choose most related studies and does not reflect any comparison or objective interpretation of selected studies.

A.6.1 A Mapping of Research Themes to Activities in REVOLVE Framework

While the REVOLVE framework has provided a broader categorisation of research, some observations and interpretation of the results suggested an explicit mapping among the identified research themes and the activities of REVOLVE framework. Figure A.2 provides a mapping of the framework activities and the identified research themes to classify and compare application and acquisition of architecture evolution reuse knowledge from Chapter 3.

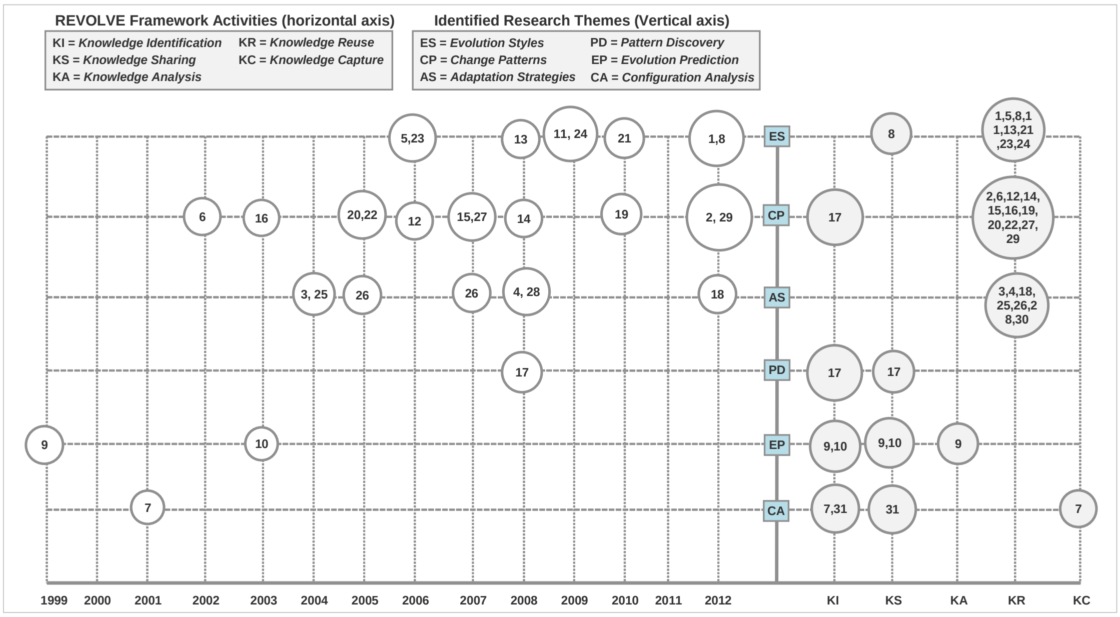


Figure A.2: Study Mapping for Research Themes, REVOLVE Activities and Publication Fora.

In this section, an iterative mapping process has been employed to present the identified research themes and to provide an answer to the first research question (SR-Q1). The map as a bubble plot is depicted in Figure A.2 to map research themes to activities of REVOLVE based on

- 5 activities of the REVOLVE framework along the horizontal axis.
- 6 identified research themes along the vertical axis.

The circles on right axis in Figure A.2 represent a mapping between framework activities and identified research themes for a study reference (e.g., '8' represents 'S8' in the Appendix list of selected studies). Alternatively, the circles on left axis represent a publication map (providing a temporal distribution, 1999 to 2012) for framework activities and identified research themes.

For example in Figure A.2, the bubble at right-axis and at the intersection of "research theme" change pattern (CP) and "framework activity" knowledge reuse (KR) represents the studies [S2, S6, S12, S14, S15, S16, S19, S20, S22, S27, S29] that support change patterns to apply reuse knowledge in ACSE. Alternatively, the bubble at the left-axis that intersects "CP" and 2012 represents the studies [Lehman 1996, Li 2012] published in 2012 and focus on change patterns. The relative size of the bubble indicates the total number of studies (the bigger the size, the more studies that theme represents).

A.7 Threats to Validity of SLR

Although the observations and results of systematic reviews are considered to be reliable [Petticrew 2008, Zhang 2012], this type of review work has its own limitations that should be considered [Garg 2008]. We discuss the each of the validity threats associated to different steps in our SLR.

A.7.1 Threats to the Identification of Primary Studies

In general, the *external validity* and *construct validity* are strong for systematic reviews [Brereton 2007]. In our search strategies, the key idea was to retrieve as much as possible of the available literature to avoid any possible bias. Another critical challenge in addressing these threats was to determine the scope of our study, since the notion of reuse knowledge refers to different communities including *software architecture*, *software product-lines* and *self-adaptive software* which use different terminologies. Therefore, to cover all and avoid bias, we searched for common terms and combined them in our search string. While this approach decreases the bias, it also significantly increases the search work. To identify relevant studies and ensure the process of selection was unbiased, a review protocol was developed.

A.7.2 Threats to Selection and Data extraction Consistency

We have identified a lack of consistent terminologies for reuse knowledge. This poses difficulties for the composition of the search queries and the inclusion/exclusion criteria. Such difficulties led us to analyse the terms concerning reuse knowledge that were found on the selected studies. However, since the notion of "reuse knowledge" is used in numerous studies, but we are specifically concerned with "architecture (-based) evolution reuse knowledge", we had to exclude a majority of retrieved studies that affected the low precision of our search. In addition, we performed a quality assessment (**Section A.5** for details) on the studies to ensure that the identified findings and implications came from credible sources.

A.7.3 Threats to Data Synthesis and Results

The threat to the reliability of results is mitigated as far as possible by involving multiple researchers, having a unified scheme, and several steps where the scheme and process were piloted and externally evaluated. Although as a general practice, we were determined to use the guidelines provided in [Brereton 2007] to perform the review, we had deviations from their procedures.

Appendix B

Case Studies for Architecture Change Mining and Change Execution Processes

Contents

B.1 Architecture Evolution Case Studies

In software engineering research, case study based approaches facilitate designing research process and evaluating results by ensuring that data is collected and analysed in a systematic and scenario-driven environment [Flyvbjerg 2006].

B.1.1 Case Studies Selection

We present the details of selected case studies that include scenarios of architecture-centric evolution about i) Electronic Bill Presentment and Payment (EBPP) System [EBPPCaseStudy], ii) *3-in-1 Phone System* [3-in-1 Phone System 1999] and iii) *Client Server Appointment* (CS-AS) System [Rosa 2004]. We also explain how these case studies help us to design, refine and evaluate the change mining and change execution processes. It is also vital to mention that while following CAR methodology [Davison 2004], case study selection is a critical process that requires a careful selection of case study data that is subject to analytical evaluation. The key characteristics for selecting case studies in software engineering are detailed in [Davison 2004]. The architecture evolution case study of EBPP is utilised as a running example throughout this thesis to elaborate on framework processes and activities along with scenario-based evaluation of results. We summarise the objectives with a list of questions for case study selection:

- *What are the primary objective(s) of the research investigation?*
- *What are the primary subject(s) of the research investigation?*
- *How selected case studies are mapped to the framework processes?*

These objectives are already outlined as research hypothesis and questions (in Chapter 1). Here, we aim to select the case studies that help us to analyse architecture evolution and also to support change execution on architecture models.

Framework Process	**Selected Case Study**	**Intent of Investigation**
Architecture Change Mining	Electronic Bill Presentment and Payment 3-in-1 Phone System	Change Classification and Operational Dependencies Discover Architecture Change Patterns
Architecture Change Execution	Client-Server Appointment System	Evolution of Architecture Pattern-based Reuse in Architecture Change Execution

Table B.1: Selected Case Studies along with the Intent of Case-study based Investigation.

The case studies include architectural evolution case for an i) Electronic Bill Presentment and Payment System (EBPP) and ii) 3-in-1 Phone system. We have selected these case studies based on **availability** and **completeness** of architecture evolution data. The intent of investigation during architecture change mining is to analyse a fine granular representation of architecture change instances that accumulate over-time to represent architecture evolution history.

The case study include the architecture evolution case of a client-server appointment system (CS-AS) [Rosa 2004]. We have selected these case study based on **availability** of i) architectural descriptions and ii) evolution scenarios. The primary intent of change execution is to support pattern-driven, reusable change execution to support architecture evolution.

B.2 Case Studies for Architecture Change Mining Process

In this section, we introduce the case studies used in the architecture change mining process to classify architecture change operations and change patterns discovery. We use the i) Electronic Bill Presentment (EBPP) [EBPPCaseStudy] and ii) 3-in-1 Telephonic System [3-in-1 Phone System 1999] case studies. Architectural changes from these case studies are captured in the change log for log-based investigation of architecture evolution[1]. We have selected these two case studies because of the availability and completeness of log data. In addition, the adequacy of change log data refers to a systematic structuring of the log that ensures availability and completeness of information. The granularity of change representation ensures completeness of syntax and semantics for recorded change instances.

In Section B, we have already justified the rationale for selection of these case studies. Here we primarily focus on presenting:

- What is a component and connector architectural view for EBPP and 3-in-1 Telephonic System case studies in Section B.2.1 and Section B.2.2.
- How changes from these two case studies are captured in the change logs for pattern discovery in Section B.2.3.

Data about architectural changes for these two case studies provide a source of knowledge to classify architecture change operationalisation (in Chapter 6) and change pattern discovery (in Chapter 7).

B.2.1 Case Study I - Architectural View for EBPP Case Study

A high-level component and connector view for EBPP is presented in Figure B.1. For illustrative reasons, we abstract the details about data store (DS) and user interface (UI) layers and focus on architectural layers modelling components and connectors using implicit configurations []. These configurations represent Metering (to provide meter information for customer's consumption), Billing (to handle customer billing), and Payment (to manage customer payments corresponding to the billing amount). We are interested in component, connectors and the interaction (messaging) that exists among the components.

[1] Each individual architectural change is captured in the log file as the basis for pattern discovery from change logs provided here: `http://ahmadaakash.wix.com/aakash#!changelogdata/c22ju`

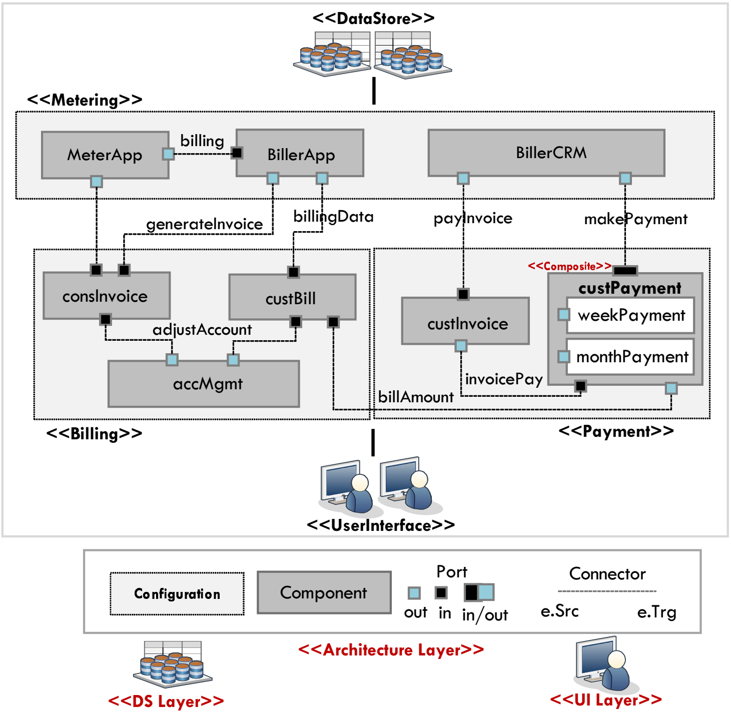


Figure B.1: Architectural View for EBPP (before Evolution).

- **Component (CMP)** represents the first class entities as computational elements or data stores of the EBPP architecture model, illustrated in Figure B.1. Component type classification is:

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

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

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

B.2.2 Case Study II - Architectural View for 3-in-1 Telephonic System Case Study

The component and connector architectural view for the 3-in-1 Telephonic System is presented in Figure B.2. The architecture consists of the four components namely Receiver, NetworkProtocol, MultimediaPlayback, and 3-in-1TelephoneHandset. The architectural changes are captured in the log as detailed in Section B.2.

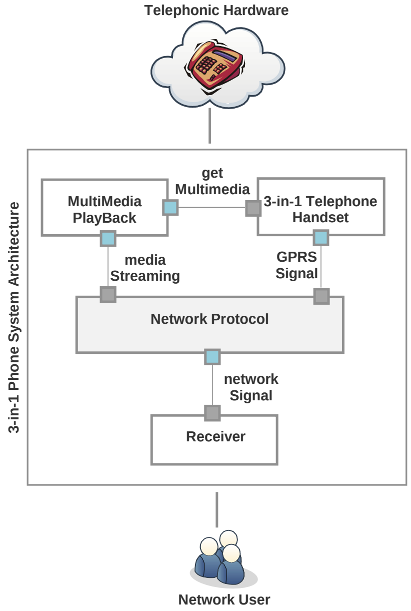


Figure B.2: Architectural Overview for 3-in-1 Phone System.

B.2.3 Capturing Architectural Changes for EBPP in Log

We look at an evolution scenario to exemplify how individual changes are captured in the change log. In order to illustrate this, we present i) the description of the evolution scenario, ii) architectural changes applied on the source model to obtain the target or evolved model and iii) recording the changes in the log.

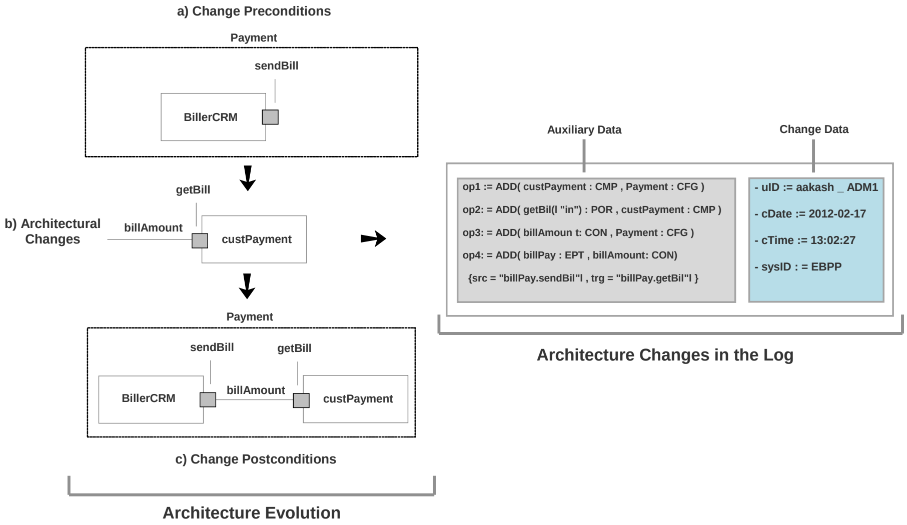
- *Evolution Scenario* The architecture evolution scenario in Figure B.3 aims to add a new component and connect it to an existing one in the architecture model. More specifically the scenario implies:

  'Add a new component custPayment *along with its port* custBill *in an existing configuration* Payment*. The newly added component* custPayment *must be connected to* BillerCRM *component with*

addition of a connector billAmount'

- *Architectural Changes* In order to implement these changes, architectural changes must be applied on the source architecture model to achieve its evolution, i.e. addition of a new component. The source architecture model (before evolution) is presented in Figure B.3 a), architectural change on the source model are presented in Figure B.3 b), while the target or evolved model (after evolution) as a consequence of change execution is presented in Figure B.3 c).

- *Capturing Architectural Changes in the Log* During architecture change execution, we must capture each individual change in the log. Change log provides a central and updated repository for architectural changes. The example in Figure B.3 represents instances of architectural change as a sequence of operations that enable addition of a new component custPayment along with its port custBill and corresponding operation getBill (op1, op2, op3). The newly added component custPayment is connected to BillerCRM with addition of a connector billAmount. It provides endpoint binding (op4) among the operations of BillerCRM and custPayment inside Payment configuration. Once sequential architectural changes are captured, change log data is classified as *Auxiliary Data (AD)* and *Change Data (CD)*, in Figure B.3. Details about change data and auxiliary data are provided in **Chapter 5**.



```
op1 := ADD( custPayment : CMP , Payment : CFG )
op2: = ADD( getBil(l "in") : POR , custPayment : CMP )
op3: = ADD( billAmoun t: CON , Payment : CFG )
op4: = ADD( billPay : EPT , billAmount: CON)
  {src = "billPay.sendBil"l , trg = "billPay.getBil"l }
```

```
- uID := aakash _ ADM1
- cDate := 2012-02-17
- cTime := 13:02:27
- sysID : = EBPP
```

Figure B.3: An Overview of Capturing architectural Changes in the Log.

B.3 Case Study for Architecture Change Execution Process

An overview of the source architecture for a Peer 2 Peer Appointment System [Rosa 2004] is presented in Figure B.5. We utilise this case studies and its evolution scenarios to evaluate pattern-based reuse of architecture evolution in Chapter 9. The architectural view consists of two configurations namely Client and AppointmentSchedule. In the source architecture, the clients make a request for appointment scheduling. The configurations consists of:

- **Components (CMP)** are the computational elements contain ports. In Figure two the components and AppointmentClients and ClientAuthentication as *atomic components*. In addition, AppointmentsSchedule is a *composite component* (composed of ClientAuthentication Component)

- **Connectors (CON)** enable interconnection among the architectural components. The only connector in the source architecture is getAppointments in Figure B.5.

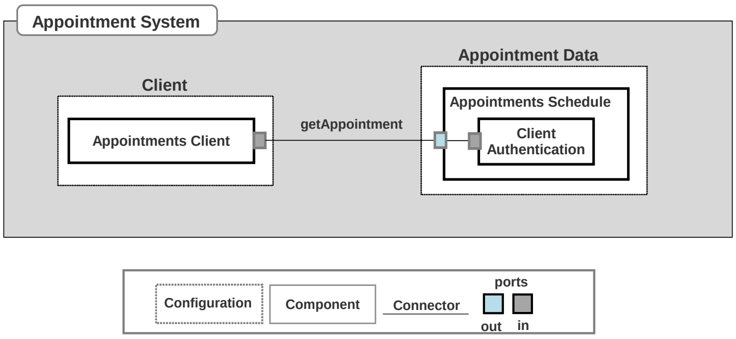


Figure B.4: An Overview of the Architecture for Peer 2 Peer Appointment System.

B.3.1 Architectural Description and Evolution Scenario

We provide architecture descriptions using attributed typed graphs for graph-based modelling of architecture elements. Details about graph-based description of architecture model is provided in **Chapter 2**.

In the following we discuss a sample evolution scenario that causes evolution in the existing architecture as illustrated in Figure B.5. The evolution Scenario is presented in Figure B.5.

The architecture is modified by creating the ClientRegistration component (atomic component) in Appointment Server (composite component) and a connector (register).

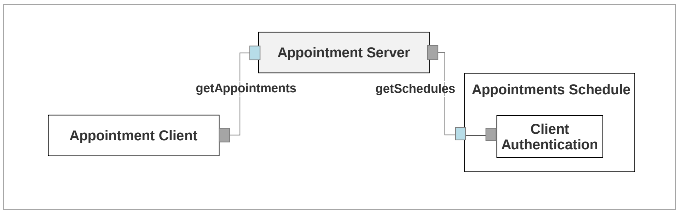


Figure B.5: An Architecture Evolution Scenario for Peer 2 Peer Appointment System.

Additional details about the architecture evolution scenarios and the target/evolved architecture model are presented in **Chapter 9**.

Appendix C

Change Log Graph for Pattern Discovery

Contents

C.1 Architecture Change Log Data

As discussed in Chapter 5, the availability of the change log data is fundamental to change operation definition (Chapter 6), pattern discovery (Chapter 7) and pattern language composition (Chapter 8). The change log data comprises of individual architectural change operations (atomic changes cf. Chapter 6) to add, remove and modify the architectural components and connectors for EBPP [EBPPCaseStudy] and 3-in-1 Phone System [3-in-1 Phone System 1999] case studies. The changes from these case studies are captured into a **change log file** [1]. Additional details about the number of change operations and the affected architecture elements as represented in the change log are provided in **Appendix E** - experimental setup for framework evaluation. Before we discuss the change log data, we also highlight the assumptions that are considered about the change log data. In the following we only provide a snippet (10 architectural changes) of the change log data for illustrative reasons, while full log data is provided on the link in the footnote.

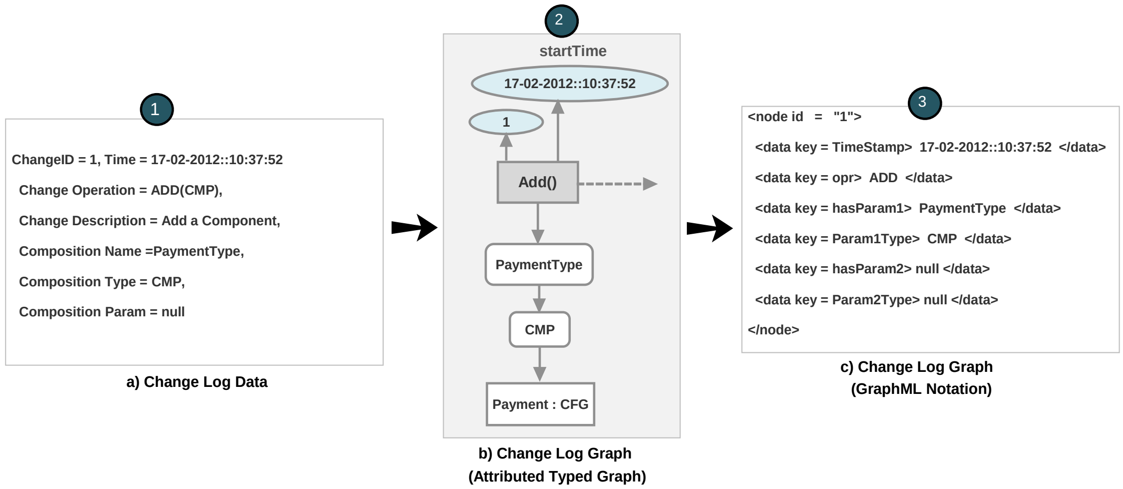
[1] Each individual architectural change is captured in the log file as the basis for pattern discovery from change logs provided here: `http://ahmadaakash.wix.com/aakash#!changelogdata/c22ju`

```
 ChangeID = 1, Change Operation = ADD(CMP), Change Description = Add a Component
Composition Name = customerBiller, Composition Type = CMP, Composition Param = null
Composite Name = porcustomerBiller, Composite Type = POR

ChangeID = 2, Change Operation = ADD(CON), Change Description = Add a Connector
Composition Name = customerBill, Composition Type = CON, Composition
Param = (customerBiller, customerPayment) : CMP
Composite Name = eptCustomerBilling, Composite Type = EPT

ChangeID = 3, Change Operation = ADD(CMP), Change Description = Add a Component
Composition Name = customerBillingApp, Composition Type = CMP, Composition Param = null
Composite Name = IbillingApp, Composite Type = POR
Composite Name = IcustomerConsumption, Composite Type = POR

ChangeID = 4, Change Operation = ADD(CON), Change Description = Add a Connector
Composition Name = customerBilling, Composition Type = CON, Composition
Param = (customerBillingApp, customerPORo) : CMP
Composite Name = BcustomerBillPORo, Composite Type = EPT

ChangeID = 5, Change Operation = ADD(CON), Change Description = Add a Connector
Composition Name = customerDebt, Composition Type = CON, Composition
Param = (customerBillingApp, debtPORo) : CMP
Composite Name = BcustomerDebt, Composite Type = EPT

ChangeID = 6, Change Operation = ADD(CON), Change Description = Add a Connector
Composition Name = customerInvoice, Composition Type = CON, Composition
Param = (customerBillingApp, invoicePORo) : CMP
Composite Name = BcustomerInvoice, Composite Type = EPT

ChangeID = 7, Change Operation = ADD(CON), Change Description = Add a Connector
Composition Name = customerInvoiceData, Composition Type = CON, Composition
Param = (customerBillingApp, invoicePORo) : CMP
Composite Name = BcustomerInvoiceData, Composite Type = EPT

ChangeID = 8, Change Operation = REM(CON), Change Description = Remove a Connector
Composition Name = customerBill, Composition Type = CON, Composition
Param = (customerBiller, customerPORo) : CMP
Composite Name = BcustomerBill, Composite Type = EPT

ChangeID = 9, Change Operation = ADD(CMP), Change Description = Add a Component
Composition Name = generateCustomerBill, Composition Type = CMP, Composition Param = null
Composite Name = IgenerateBill, Composite Type = POR
Composite Name = IcustomerInvoice, Composite Type = POR

ChangeID = 10, Change Operation = ADD(CON), Change Description = Add a Connector
Composition Name = generateBill, Composition Type = CON, Composition
Param = (customerBiller, generateCustomerBill) : CMP
Composite Name = BcustomerBilling, Composite Type = EPT
```

- Atomic Changes in the Log - We assume that all the architectural changes in the log represents atomic change operations on architecture elements [Ahmad 2012b].
- Sequential Changes in the Log Our assumption is that the architectural changes are applied in a sequential fashion. If there exist any parallel changes that are represented as a sequence [Buckley 2005].
- Completeness of the Log Data - We also assume that no changes from the case studies re omitted or skipped and change log data is complete [Yu 2009].

C.2 Converting Log Data into a Change Log Graph

- As detailed in Chapter 5, we model change log data as an attributed typed graph [Ehrig 2006] in order to exploit sub-graph mining for pattern discovery. In this appendix, we provide a sample of the change log graph (20 change operations). In Figure C.1, we illustrate



ChangeID = 1, Time = 17-02-2012::10:37:52
Change Operation = ADD(CMP),
Change Description = Add a Component,
Composition Name =PaymentType,
Composition Type = CMP,
Composition Param = null

a) Change Log Data

b) Change Log Graph
(Attributed Typed Graph)

```
<node id  =  "1">
  <data key = TimeStamp>  17-02-2012::10:37:52  </data>
  <data key = opr>  ADD  </data>
  <data key = hasParam1>  PaymentType  </data>
  <data key = Param1Type>  CMP  </data>
  <data key = hasParam2> null </data>
  <data key = Param2Type> null </data>
</node>
```

c) Change Log Graph
(GraphML Notation)

Figure C.1: Graph-based Representation of the Change Log Data.

- *Change Log Data:* In Figure C.1 a), presents an individual change operation [Ahmad 2012b] representing an operation for addition of a component PaymentType
- *Attributed Graph-based Representation of Log Data:* In Figure C.1 b), we present an attributed typed graph-based [Ehrig 2006] representation for an individual change operation. The change operation (Add()) is represented as the graph node, while the parameters of the operation (PaymentType $\in$ CMP) is represented as node attribute. The sequence among the change operation is maintained with graph edges.

- *GraphML-based Representation of Log Data:* Finally, in Figure C.1 c), we present a GraphML [Brandes 2002a] based representation of the change log data. It contains an XML-based representation of the log graph for an automated manipulation of log data and pattern discovery.

In Chapter 5, we have discussed the role of the prototype to convert the log data (provided as input) into a change log graph (as the output).

C.3 Sample Log Graph for Change Pattern Discovery

A Sample Log Graph using GraphML representation is presented as follows.

```
<?xml version = "1.0" encoding = "UTF–8"?>

<graphml xmlns = "http://graphml.graphdrawing.org/xmlns"

 xmlns:xsi = "http://www.w3.org/2001/

   XMLSchema–instance" xsi:schemaLocation

   = "http://graphml.graphdrawing.org/xmlns

 http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

 <!–– Graph generated by sones GraphAPI GraphMLWriter ––>

 <key id = "opr" for = "node" attr.name = "opr" attr.type = "string">    </key>

 <key id = "hasParam1" for = "node" attr.name = "hasParam1" attr.type = "string">    </key>

 <key id = "Param1Type" for = "node" attr.name = "Param1Type" attr.type = "string">    </key>

 <key id = "hasParam2" for = "node" attr.name = "hasParam2" attr.type = "string">    </key>

 <key id = "Param2Type" for = "node" attr.name = "Param2Type" attr.type = "string">    </key>

 <key id = "seq" for = "edge" attr.name = "type" attr.type = "composition">    </key>


 <graph id = "LogGraph" edgedefault = "directed">


 <node id = "1">
 <data key = opr> ADD </data>

 <data key = hasParam1>  customerBiller  </data>

 <data key = Param1Type>  CMP  </data>

<data key = hasParam2> null </data>

<data key = Param2Type> null </data>




<node id  =  "2">

<data key = opr> ADD </data>

<data key = hasParam1> customerBill </data>

<data key = Param1Type> CON </data>

<data key = hasParam2> customerBiller , customerInfo </data>

<data key = Param2Type> CMP </data>




<node id  =  "3">

<data key = opr> ADD </data>

<data key = hasParam1> customerBillingApp </data>

<data key = Param1Type> CMP </data>

<data key = hasParam2> null </data>

<data key = Param2Type> null </data>




<node id  =  "4">

<data key = opr> ADD </data>

<data key = hasParam1> customerBilling </data>

<data key = Param1Type> CON </data>

<data key = hasParam2> customerBillingApp , customerInfo </data>

<data key = Param2Type> CMP </data>
```

```

<node id  =  "5">

<data key = opr> ADD </data>

<data key = hasParam1> customerDebt </data>

<data key = Param1Type> CON </data>

<data key = hasParam2> customerBillingApp , debtInfo </data>

<data key = Param2Type> CMP </data>


<edge id =  "e1" source = "1" target = "2">
</edge>

<edge id =  "e2" source = "2" target = "3">
</edge>

<edge id =  "e3" source = "3" target = "4">
</edge>

<edge id =  "e4" source = "4" target = "5">
</edge>


</graph>

</graphml>
```

Appendix D

Source Code and Discovered Change Patterns from Log

Contents

In this appendix, we present the Java source code we developed for pattern discover from architecture change logs in Section D.5. We also present an overview of the discovered pattern instances in Section D.1

D.1 Source Code for Pattern Discovery from Logs

We present the source code[1] for pattern discovery in the following. The source code listed here only present the core executable java code that is associated to the prototype GPride already presented in Chapter 7. We provide the code for:

[1] All Java source code files for the prototype GPride are available at: `http://ahmadaakash.wix.com/aakash/GPrideCode`

- *Pattern Candidate Generation* enables generation of a list of pattern candidates.

- *Pattern Candidate Validation* enables validation of the generated candidates. Candidate validation ensures that each individual candidate must preserved the structural integrity of architecture model.

- *Pattern Matching* enables matching the generated candidates to discover patterns. A generated candidate is a pattern if it satisfies the specified pattern frequency threshold.

Additional technical details about pattern discovery process and the underlying algorithms are provided in Chapter 7.

```java

/**************************************************
 * Java Source Code for Candidate Generation

 * @Author: Aakash Ahmad
 * @Date: 08 - 17 - 2012
**************************************************/
import GraphMLDemo.node;
import java.io.Serializable;
import java.util.*;

public class Candidate implements Serializable{
  public class Candidate
  {
        public boolean bIsValid;
        public ArrayList<node> NodesList;

        public Candidate()
      {
              this.bIsValid = true;
              this.NodesList = new ArrayList<node>();
        }

        public void AddNode(node oNode)
      {
              this.NodesList.add(oNode);
        }

      public static boolean MAP ExactlyContainsAll(Map<Candidate, Integer>
                                                   ExactMatchingCandidates, Candidate C2)
      {
        for (Map.Entry<Candidate, Integer> entry : ExactMatchingCandidates.entrySet())
        {
```

```
    Candidate C1 = entry.getKey();
     if (C1.IsExactlyEqual(C2))
        return true;
  }
    return false;
}

  public boolean IsExactlyEqual(Candidate c2)
{
  boolean bIsNodeLengthSame = this.NodesList.size() == c2.NodesList.size(),
  bIsCandidate1Valid = this.bIsValid,
  bIsCandidate2Valid = c2.bIsValid;

  if (bIsNodeLengthSame && bIsCandidate1Valid && bIsCandidate2Valid)
  {
      boolean bIsExactlyMatched = true;
      for (int nNodeIndex = 0; nNodeIndex < this.NodesList.size(); nNodeIndex++)
      {
       if (!(this.NodesList.get(nNodeIndex).equals(c2.NodesList.get(nNodeIndex))))
       {
          bIsExactlyMatched = false;
          break;
       }
      }
      return bIsExactlyMatched;
  }
      return false;
   }

 public static boolean MAP InExactlyContainsAll(Map<Candidate, Integer>
                                              InExactMatchingCandidates, Candidate C2)
 {
  for (Map.Entry<Candidate, Integer> entry : InExactMatchingCandidates.entrySet())
  {
    Candidate C1 = entry.getKey();
    if (C1.IsInExactlyEqual(C2))
      return true;
  }
      return false;
 }

 public boolean IsInExactlyEqual(Candidate c2)
 {
   boolean bIsNodeLengthSame = this.NodesList.size() == c2.NodesList.size(),
   bIsCandidate1Valid = this.bIsValid,
   bIsCandidate2Valid = c2.bIsValid;

   if (bIsNodeLengthSame && bIsCandidate1Valid && bIsCandidate2Valid)
   {
    boolean bIsNodeIndexMisMatch = false;
    ArrayList<Integer> C2MatchingIndexes = new ArrayList<Integer>();
    for (int Candidate1NodeIndex =0; Candidate1NodeIndex< this.NodesList.size();
                  Candidate1NodeIndex++)
    {
      node C1CurrNode = this.NodesList.get(Candidate1NodeIndex);
      boolean bIsNodeMatch = false;

      for (int Candidate2NodeIndex =0; Candidate2NodeIndex< c2.NodesList.size();
                      Candidate2NodeIndex++)
      {
        node C2CurrNode = c2.NodesList.get(Candidate2NodeIndex);
        if ((C1CurrNode.equals(C2CurrNode)))
        {
          if (!(C2MatchingIndexes.contains(Candidate2NodeIndex)))
          {
            C2MatchingIndexes.add(Candidate2NodeIndex);
            bIsNodeMatch = true;
            if (Candidate1NodeIndex != Candidate2NodeIndex)
              bIsNodeIndexMisMatch = true;
              break;
           }
         }
       }
       if (!bIsNodeMatch)
         return false;
     }
   if (bIsNodeIndexMisMatch)
    return true;
  }
   return false;
  }

public String toString()
 {
   String sNewLine = System.getProperty("line.separator");
   String sTab = "      ";
   StringBuffer buffer = new StringBuffer();

   buffer.append(sNewLine + "Total Nodes: ");
   buffer.append(this.NodesList.size());

   buffer.append(sNewLine + "Generic Form: ");
   buffer.append(this.GetCandidateGenericForm());

   buffer.append(sNewLine + "Nodes Detail:"+ sNewLine);

         for (int nNodeIndex = 0; nNodeIndex < this.NodesList.size(); nNodeIndex++)
           {
                node CurrNode = this.NodesList.get(nNodeIndex);
                buffer.append(sTab + "(");
                buffer.append("Node Id: " + CurrNode.getValue());
                buffer.append(" Operator: " + CurrNode.getOperator());
                buffer.append(" Param1: " + CurrNode.getParam1());
                buffer.append(" Param1Type: " + CurrNode.getParam1Type());
                buffer.append(" Param2: " + CurrNode.getParam2());
                buffer.append(" Param2Type: " + CurrNode.getParam2Type());
                buffer.append(")" + sNewLine);
            }
            return buffer.toString();
      }

      public String GetCandidateGenericForm()
      {
        String sGenericForm ="";
        for (int NodeIndex =0; NodeIndex < this.NodesList.size(); NodeIndex++)
        {
          node CurrNode = this.NodesList.get(NodeIndex);
          sGenericForm += CurrNode.getOperator() + " (" + CurrNode.getParam1Type() + ") ";
        }
        return sGenericForm;
      }
////////////////////////////////////////////////////////////////////////////////////
//  Extra Code starts from this point
////////////////////////////////////////////////////////////////////////////////////

    public void DisplayCandidate() //not required now
    {
      for (int nNodeIndex = 0; nNodeIndex < this.NodesList.size(); nNodeIndex++)
      {
        node CurrNode = this.NodesList.get(nNodeIndex);
        System.out.print("(");
        System.out.print("Node Id: " + CurrNode.getValue());
        System.out.print(" Operator: " + CurrNode.getOperator());
        System.out.print(" Param1: " + CurrNode.getParam1());
        System.out.print(" Param1Type: " + CurrNode.getParam1Type());
        System.out.print(" Param2: " + CurrNode.getParam2());
        System.out.print(" Param2Type: " + CurrNode.getParam2Type());
        System.out.print("), ");
      }
     System.out.println("");
      }

    public static void DisplayAllCandidates(List<Candidate> CandidatesList)
    {
     Candidate TempCandidate = new Candidate();
     for (int Index = 0; Index < CandidatesList.size(); Index++)
     {
       TempCandidate = CandidatesList.get(Index);
       TempCandidate.DisplayCandidate();
     }
      }

   public boolean IsInExactlyEqual(Candidate c2)
   {
     boolean bIsNodeLengthSame = this.NodesList.size() == c2.NodesList.size(),
     bIsCandidate1Valid = this.bIsValid,
     bIsCandidate2Valid = c2.bIsValid;

     if (bIsNodeLengthSame && bIsCandidate1Valid && bIsCandidate2Valid)
     {
        boolean bIsAllNodeLocationMismatch = false, bIsNodeMisMatch = false;
        for (int Candidate1NodeIndex =0; Candidate1NodeIndex< this.NodesList.size();
                      Candidate1NodeIndex++)
        {
         if (c2.NodesList.contains(this.NodesList.get(Candidate1NodeIndex)))
         {
           if (Candidate1NodeIndex != c2.NodesList.indexOf(this.NodesList.
                          get(Candidate1NodeIndex)))
           {
            bIsAllNodeLocationMismatch = true;
           }
         }
         else
        {
         bIsNodeMisMatch = true;
         break;
        }
       }
      if (bIsAllNodeLocationMismatch && (!bIsNodeMisMatch))
        return true;
    }
    return false;
}

public boolean equals(Candidate c2)
 {
   int nC1Size =this.NodesList.size(),
   nC2Size = c2.NodesList.size();
    if (nC1Size != nC2Size)
     return false;
    if (this.NodesList.containsAll(c2.NodesList))
     return true;
    for (int nNodeIndex = 0; nNodeIndex< nC1Size; nNodeIndex++)
    {
     node C1CurrNode = this.NodesList.get(nNodeIndex),
     C2CurrNode = c2.NodesList.get(nNodeIndex);
```

```
        if (!(C1CurrNode.equals(C2CurrNode)))
          return false;
       }
        return false;
}

}
```

```

/*************************************************
 * Java Source Code for Candidate Validation

 * @Author: Aakash Ahmad
 * @Date: 09 − 02 − 2012
*************************************************/

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;

public class CandidateValidator {

 public static void ValidateAllCandidates(Map<Candidate, Integer> CandidatesList)
 {
   for (Map.Entry<Candidate, Integer> entry : CandidatesList.entrySet())
   {
      Candidate CurrCandidate = entry.getKey();
      if (!ValidateCandidate(CurrCandidate))
      {
        CandidatesList.remove(entry.getKey());
      }
   }
}

 private static boolean ValidateCandidate(Candidate c)
 {
     int nC1Size =this.NodesList.size(),
   nC2Size = c2.NodesList.size();
    if (nC1Size != nC2Size)
     return false;
    if (this.NodesList.containsAll(c2.NodesList))
     return true;
    for (int nNodeIndex = 0; nNodeIndex< nC1Size; nNodeIndex++)
    {
     node C1CurrNode = this.NodesList.get(nNodeIndex),
     C2CurrNode = c2.NodesList.get(nNodeIndex);
```

```java
        if (!(C1CurrNode.equals(C2CurrNode)))
          return false;
       }
        return false;
 }

 }
 }
```

```java

/**************************************************
 * Java Source Code for Pattern Matching

 * @Author: Aakash Ahmad
 * @Date: 09 - 14 - 2012
**************************************************/
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.TreeMap;
import GraphMLDemo.node;


public class PatternMatcher {
public static void IdentifyExactAndInExactMatch
            (Map<Candidate, Integer> CandidatesList, Map graph, Map<Candidate, Integer>
             ExactMatchingCandidates, Map<Candidate, Integer>
             InExactMatchingCandidates, int nExactMatchFreqThreshold,
             int nInExactMatchFreqThreshold, boolean IdentifyExactMatch,
             boolean IdentifyInExactMatch)
{
        ExactMatchingCandidates.clear();
        InExactMatchingCandidates.clear();
    for (Map.Entry<Candidate, Integer> entry : CandidatesList.entrySet())
    {
     Candidate CurrCandidate = entry.getKey();
     int nPatternLength = CurrCandidate.NodesList.size(),
     nExactMatchFrequency = 0,
     nInExactMatchFrequency = 0;

      for (int nMapIndex=0; nMapIndex ≤ (graph.size()- nPatternLength) ; nMapIndex++)
        {
         Candidate GraphCurrCandidate = new Candidate();
         for (int nItr=0; nItr < nPatternLength; nItr++)
         {
             GraphCurrCandidate.AddNode((node)graph.get(nItr+nMapIndex));
         }

        if (IdentifyExactMatch)
    {
     if (CurrCandidate.IsExactlyEqual(GraphCurrCandidate))
         nExactMatchFrequency++;
    }
     if (IdentifyInExactMatch)
     {
       if (CurrCandidate.IsInExactlyEqual(GraphCurrCandidate))
             nInExactMatchFrequency++;
     }
     }

     if (nExactMatchFrequency≥nExactMatchFreqThreshold)
   {
     if (!(Candidate.MAPExactlyContainsAll(ExactMatchingCandidates, CurrCandidate)))
     ExactMatchingCandidates.put(CurrCandidate, nExactMatchFrequency);
   }

        if (nInExactMatchFrequency≥nInExactMatchFreqThreshold)
    {
     if (!(Candidate.MAP InExactlyContainsAll(InExactMatchingCandidates, CurrCandidate)))
      InExactMatchingCandidates.put(CurrCandidate, nInExactMatchFrequency);
    }
  }
}

public static void DisplayMatchingCandidates(Map<Candidate, Integer> MatchingCandidates)
 {
 for (Map.Entry<Candidate, Integer> entry : MatchingCandidates.entrySet()) {
  Candidate CurrCandidate = entry.getKey();
    int value = entry.getValue();

    CurrCandidate.DisplayCandidate();
    System.out.println("Frequency: "+ value);
 }

 }
}
```

D.2 Pattern, Pattern Instance and Pattern Variant

We distinguish between a pattern, pattern instance and its variant in Figure D.1 and exemplify the distinctions with the help of the Component Mediation pattern. In addition, it is vital to mention about the pre-conditions and post-conditions of a pattern that present the architecture before and after the application of a change patterns. The pattern provides a process-based change implementation by explicitly representing the conditions before, during and after the change implementation.

- Pattern A pattern represents a generic and repeatable solution to recurring architecture evolution problems presented in Chapter 7.

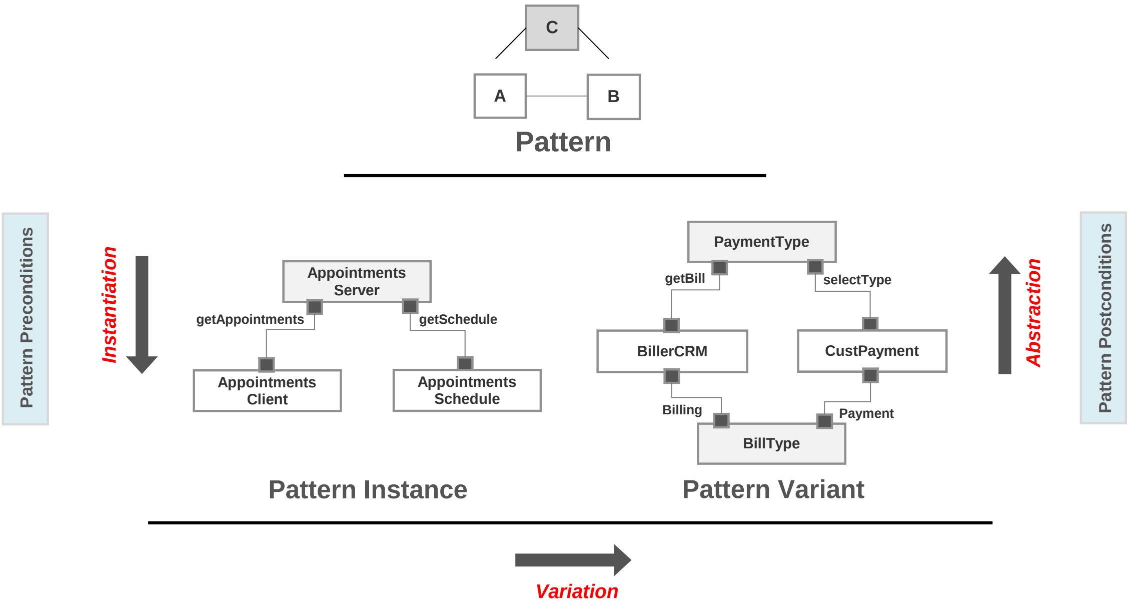


Figure D.1: An Overview of the Pattern, Pattern Instance and Pattern Variant

We defined pattern as a *'recurring constrained composition of change operationalisation on architecture elements.'*. In Figure D.1 we represent this generic solution using a simple box and arrows notation that represent the components and connectors of a pattern.

- Pattern Instance The pattern instance represents a concrete representation of the pattern. As presented in Figure D.1 the pattern instance represents the interposition of a mediator component Appointment Server component among the directly connected Appointment Client and Appointment Schedule. We have exemplified the pattern instantiation in Chapter 8.

- Pattern Variant The pattern variant represents a possible variation of the implementation of the pattern. In Figure D.1 Parallel Mediation represents the variation of the Component Mediation pattern. We have discussed the discovery of pattern variants (exact and in-exact pattern match) in Chapter 7.

D.3 Prototype Support for Change Pattern Discovery

D.3.1 Overview of the Prototype for Pattern Discovery

We have developed a prototype GPride (Graph-based Pattern Identification) presented to support automation and customisation of the pattern discovery process. A high-level view of the prototype for pattern discovery is presented in Figure D.2 in terms of the input/output, core processes and tasks. The input to the prototype GPride is a change log graph (*.GML* format) from chapter 5. The prototype has a three step process of pattern discovery including i) candidate generation, ii) candidate validation and iii) pattern matching already explained in Chapter 7. During pattern matching we discover recurring architecture change operations (from Chapter 6) as change patterns.

The user input is vital to customise the pattern discovery process. User interface for pattern discovery is presented in Figure D.3 and parametrise customisation of pattern discovery process. The output of the prototype is a list of discovered architecture change patterns.

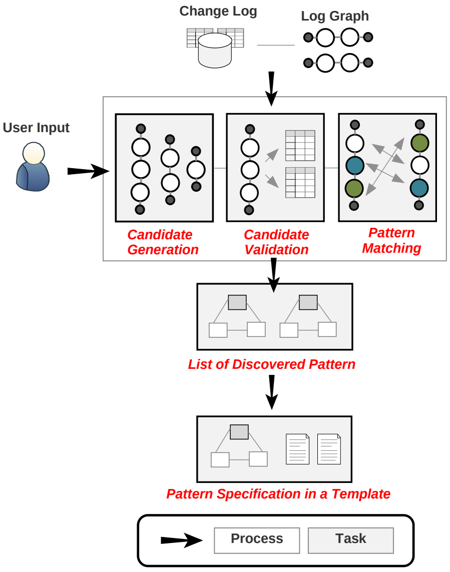


Figure D.2: An Overview of the Prototype for Change Pattern Discovery.

To support a template-based specification of change patterns, the prototype allows the user to specify each pattern in a pattern template.

D.3.2 User Interface for Pattern Discovery Prototype

In the section, we discuss the individual elements of the user interface to highlight process automation and parametrised customisation.

- **A. Log File Selection** As presented in Figure D.3, the prototype allows a user to select a specific change log graph file to start the pattern discovery process. Details about change log graph are presented in Chapter 5 and a sample log graph file in **Appendix C**.

- **B. Pattern Discovery Parameters** Pattern discovery parameters facilitate a user of the prototype to customise the pattern discovery process. The parameters for pattern discovery allow a user to specify:

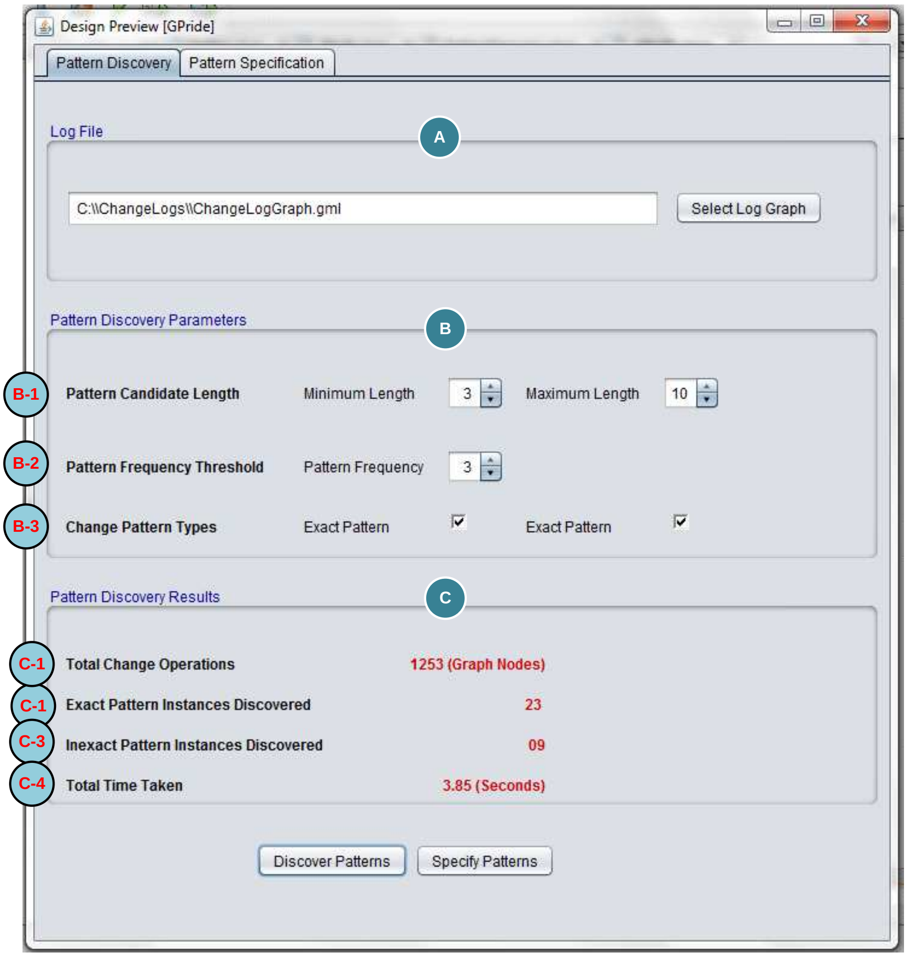
  - *Minimum and Maximum Length of the Pattern Candidate:* As already discussed in Algorithm I, a precondition to pattern discovery is generation of pattern candidates. Therefore, specifying the minimum and maximum length of the pattern candidates allows a user to specify the exact minimum (3 change operations) and exact maximum (10 change operations) length of pattern candidates in Figure D.3.

  - *Pattern Frequency Threshold:* As already discussed in Algorithm I, the user can also specify the pattern frequency threshold. It maintains (3 occurrences) a minimum frequency that must be satisfied to consider the recurring candidates as a discovered patterns.

  - *Discovery of Exact and Inexact Pattern Instances:* As already discussed in Algorithm II, the distinction between the exact and in-exact pattern instances. The prototype allows a user to specify if they want to discover both the exact (23 patterns) as well as inexact (9 pattern) instances. If the user only specifies Exact Pattern Instances, the pattern discovery process is considerable faster but it skips the inexact pattern instances.

- **C. Pattern Discovery Results** As presented in Figure D.3, it provides a summary of the results for pattern discovery process. It highlights the *total number of change operations* investigated for pattern discovery. The number of *exact as well as inexact patterns instances* discovered and the *total time taken* for pattern discovery.

  The discovered patterns need to be specified in a change pattern template.



Log File

C:\\ChangeLogs\\ChangeLogGraph.gml

Pattern Discovery Parameters

Pattern Candidate Length — Minimum Length 3 — Maximum Length 10

Pattern Frequency Threshold — Pattern Frequency 3

Change Pattern Types — Exact Pattern ☑ — Exact Pattern ☑

Pattern Discovery Results

Total Change Operations 1253 (Graph Nodes)

Exact Pattern Instances Discovered 23

Inexact Pattern Instances Discovered 09

Total Time Taken 3.85 (Seconds)

Figure D.3: Screen-shot of the Prototype for Change Pattern Discovery.

D.4 Prototype Support for Change Pattern Specification

The prototype GPride (cf. Section D.3) also allows a user to specify the change patterns in a change pattern template. In the following, we discuss the individual elements of the prototype to specify change patterns as:

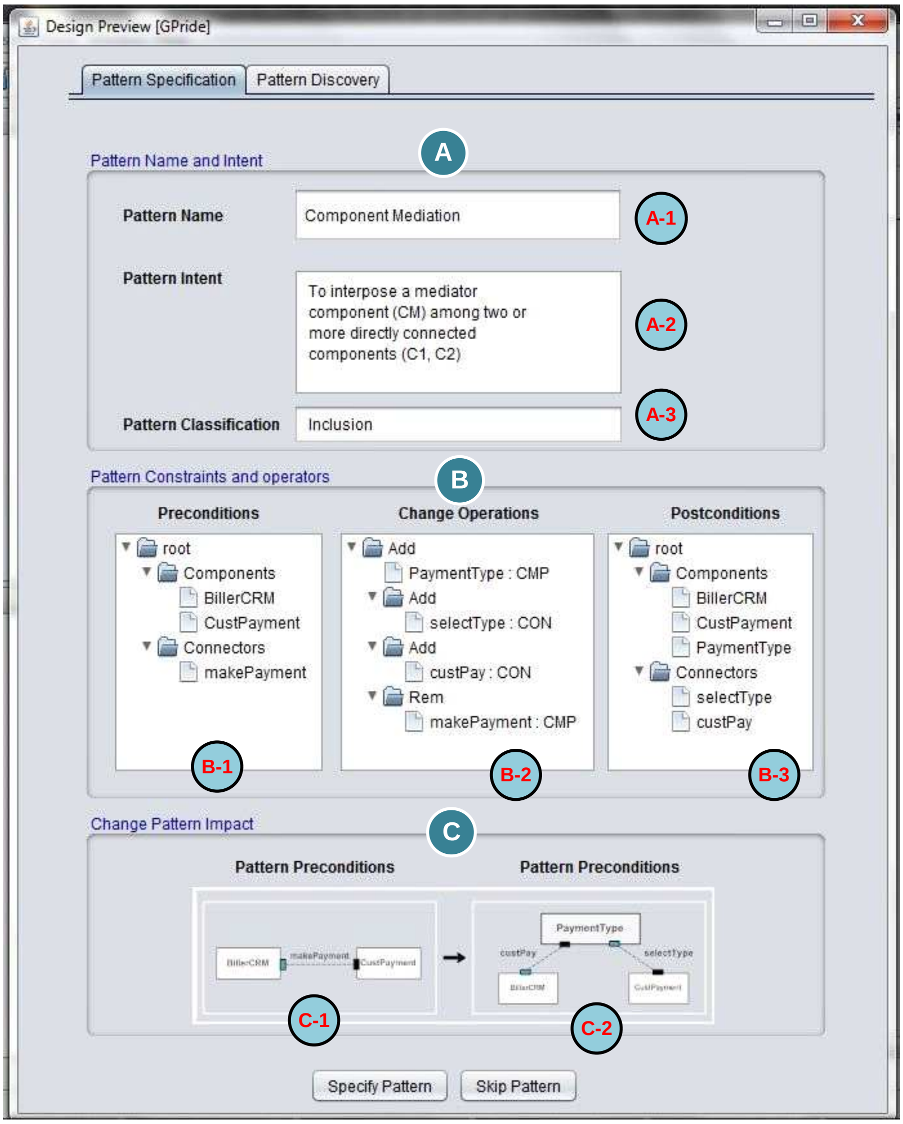
- **A. Specifying Pattern Name and Intents** As presented in Figure D.4, the prototype allows a user to specify the name, intent and classification type for a pattern. Pattern name and intent are specified by a user based on the impact of change pattern. For example, in Figure D.4 the visualisation of pattern preconditions and pattern post-conditions helps a user to identify that the pattern provides a component mediation among two directly connected components. The user also specifies the classification type of change pattern.

- **B. Pattern Constraints and Operations**

The constraints and operations are extracted from each identified pattern and presented to the user that helps to decide about the name and intent of the pattern. The constraints are presented as preconditions and post-conditions. The change operations represents individual changes on architecture model as presented in Figure D.4.

- **C. Change Pattern Impact**

  Finally, the impact of each discovered pattern is visualised to help the user to analyse the impact of change patterns before and after the application of change pattern in Figure D.4.



Design Preview [GPride]

Pattern Specification | Pattern Discovery

A

Pattern Name and Intent

Pattern Name: Component Mediation (A-1)

Pattern Intent: To interpose a mediator component (CM) among two or more directly connected components (C1, C2) (A-2)

Pattern Classification: Inclusion (A-3)

B

Pattern Constraints and operators

Preconditions (B-1)
- root
  - Components
    - BillerCRM
    - CustPayment
  - Connectors
    - makePayment

Change Operations (B-2)
- Add
  - PaymentType : CMP
  - Add
    - selectType : CON
  - Add
    - custPay : CON
  - Rem
    - makePayment : CMP

Postconditions (B-3)
- root
  - Components
    - BillerCRM
    - CustPayment
    - PaymentType
  - Connectors
    - selectType
    - custPay

C

Change Pattern Impact

Pattern Preconditions | Pattern Preconditions

Specify Pattern | Skip Pattern

Figure D.4: Screen-shot of Prototype for Change Pattern Specification.

D.5 A Catalogue of Architecture Change Patterns

A Pattern catalogue [2] refers to a collection or a repository of patterns contains patterns as repeatable solution to recurring problems in a specific domain. For example, the discovered patterns presented in this thesis (Chapter 7) represent a collection of patterns that support reuse of architecture evolution. In contrast to a pattern language, a pattern catalogue do not support relations or connections among the patterns. A pattern language goes beyond establishing a repository of patterns to support the possible relationships among the existing patterns as we have detailed in Chapter 8.

Here, we provide an overview of the discovered patterns[3]. Please note that, the pattern listing for all discovered patterns and a detailed example (Component Mediation pattern) is used to explain the template-based specification of change patterns in Chapter 7. Here in this appendix, we present the remaining patterns with the basic details specified in a pattern template. For example, to clarify we first present the Active Displacement. We provide a template-based specification for this pattern.

- **Pattern Name** Active Displacement($< C_1 : C_2 >$, $< C_1 : C_3 >$ $[C_2 : C_3]$)

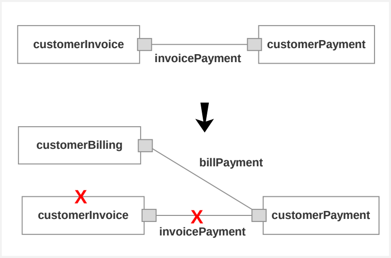


Figure D.5: Example of Active Displacement Pattern

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

- **Pattern Example** The example for the Active Displacement pattern is illustrated in Figure D.5. In this example, the customerInvoice and customerPayment components are interconnected using the connector invoicePayment. Now there is a need to replace the existing

[2] Our definition of the *Pattern Catalogue* is consistent with the view of pattern collections in the pattern community http://hillside.net/patterns/patterns-catalogue

[3] Many pattern authors utilise the term pattern thumbnails referring to overview of a pattern also known as problem/-solution mapping. Th problem refers to a specific concern or a challenge that is addressed with a specific pattern.

component customerInvoice with customerBilling. This can be achieved by applying the active displacement pattern as presented in Figure D.5.

D.5.1 Component Mediation Pattern

- **Pattern Parameters** - $ComponentMediation([C_M] < C_1, C_M, C_2 >)$
- **Pattern Intent** - To interpose a mediator component (C_M) among two or more directly connected components (C_1, C_2)

Context and Forces

- **Constraints** - Represent the architecture model before, during and after changes.

  1. *Preconditions* - C_1 and C_2 must be directly connected.
  2. *Invariants* - C_1 and C_2 must be disconnected.
  3. *Postconditions* - C_1 and C_2 must connected with C_M.

- **Change Operators** - Enables the architecture change implementation.

  1. *Add(Component)* - Add a Component C_M.
  2. *Rem(Connector)* - Remove a Connector $X_1(C_1, C_2)$.
  3. *Add(Connector)* - Add a Connector $X_2(C_1, C_M)$.
  4. *Add(Connector)* - Add a Connector $X_3(C_M, C_2)$

- **Architecture Models** - Represents the affected architecture model.

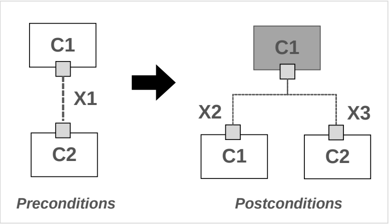


Figure D.6: Overview of the Component Mediation Pattern

D.5.2 Functional Slicing Pattern

- **Pattern Parameters** - $FunctionalSlicing([C] < C_1, C_2 >)$.
- **Pattern Intent** - To split a component (C) into two or more components (C_1, C_2) for functional decomposition of C.

Context and Forces

- **Constraints** - Represent the architecture model before, during and after changes.

  1. *Preconditions* - C already exists in the architecture.
  2. *Invariants* - N/A.
  3. *Postconditions* - C removed, C_1 and C_2 must be added.

- **Change Operators** - Enables the architecture change implementation.

  1. *Add(Component)* - Add a Component C_1 by splitting C.
  2. *Add(Component)* - Add a Component C_2 by splitting C.
  3. *Rem(Component)* - Remove a Component C.

- **Architecture Models** - Represents the affected architecture model.

D.5.3 Functional Unification Pattern

- **Pattern Parameters** - $FunctionalUnification(< C_1, C_2 > [C])$

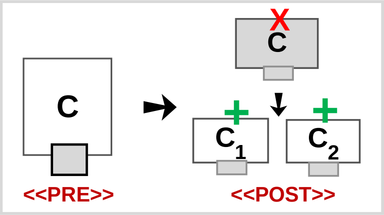


Figure D.7: Overview of the Functional Slicing Pattern

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

Context and Forces

- **Constraints** - Represent the architecture model before, during and after changes.

  1. *Preconditions* - C_1 and C_2 already exist in the architecture.
  2. *Invariants* - N/A.
  3. *Postconditions* - $C1$ and $C2$ removed, C is added.

- **Change Operators** - Enables the architecture change implementation.

  1. *Add(Component)* - Add a Component C.
  2. *Rem(Connector)* - Remove a Component C_1.
  3. *Add(Connector)* - Remove a Component C_2.

- **Architecture Models** - Represents the affected architecture model.

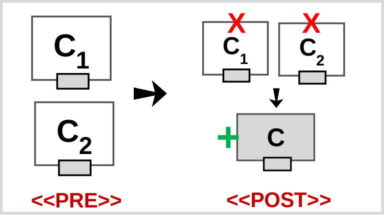


Figure D.8: Overview of the Functional Unification Pattern

D.5.4 Active Displacement Pattern

- **Pattern Parameters** - $ActiveDisplacement(<C_1 : C_2>, <C_1 : C_3 > [C_2 : C_3])$
- **Pattern Intent** - To replace an existing component (C_1) with a new component (C_3) while maintaining the interconnection with existing component (C_2).

Context and Forces

- **Constraints** - Represent the architecture model before, during and after changes.
  1. *Preconditions* - C_1 and C_2 must be directly connected.
  2. *Invariants* - C_2 exists in architecture, C_1 is removed.
  3. *Postconditions* - C_2 connected to a new component C_3.
- **Change Operators** - Enables the architecture change implementation.
  1. *Rem(Component)* - Remove a Component C_1.
  2. *Rem(Connector)* - Remove a Connector $X_1(C_1, C_2)$.
  3. *Add(Connector)* - Add a Component C_3.
  4. *Add(Connector)* - Add a Connector $X_2(C_2, C_3)$.
- **Architecture Models** - Represents the affected architecture model.

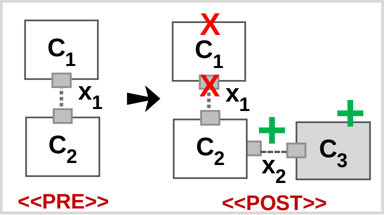


Figure D.9: Overview of the Active Displacement Pattern

D.5.5 Child Creation Pattern

- **Pattern Parameters** - $ChildCreation([C] < X_1 : C >)$
- **Pattern Intent** - To create a child component (X_1) inside an atomic component (C).

Context and Forces

- ⁎ **Constraints** - Represent the architecture model before, during and after changes.

  1. *Preconditions* - Component C is an atomic component.
  2. *Invariants* - N/A.
  3. *Postconditions* - X_1 is a child component of C (C is Composite)..

- ⁎ **Change Operators** - Enables the architecture change implementation.

  1. *Add(Component)* - Add a Component X_1.
  2. *Mov(Component)* - Move in Component X_1 inside a Component C.
  3. *Add(Connector)* - Add a Connector $X_2(C_1, C_M)$.
  4. *Add(Connector)* - Add a Connector $X_3(C_M, C_2)$

- ⁎ **Architecture Models** - Represents the affected architecture model.

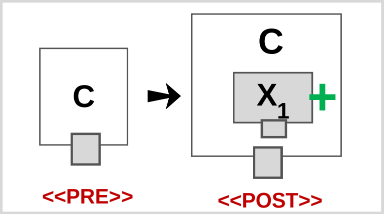


Figure D.10: Overview of the Child Creation Pattern

D.5.6 Child Adoption Pattern

- ⁎ **Pattern Parameters** - $ChildAdoptionPattern(< C_1 : X_1, C_2 >, < C_1, C_2 : X_1 >)$

- ⁎ **Pattern Intent** - To adopt a child component (X_1) from a composite component (C_1) to an atomic component (C_2).

Context and Forces

- ⁎ **Constraints** - Represent the architecture model before, during and after changes.

  1. *Preconditions* - X_1 is a child inside composite C_1.
  2. *Invariants* - X_1 is removed from C_1.

3. *Postconditions* - X_1 is added in component C_2.

- **Change Operators** - Enables the architecture change implementation.

1. *Rem(Component)* - Remove a Component X_1 from Component C_1 (C_1 is atomic).
2. *Rem(Connector)* - Add a Component X_1 into Component C_2 (C_2 is composite).

- **Architecture Models** - Represents the affected architecture model.

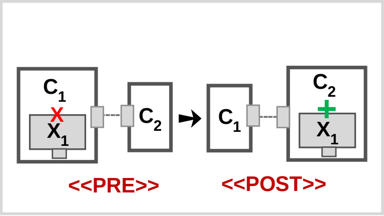


Figure D.11: Overview of the Child Adoption Pattern

D.5.7 Child Swapping Pattern

- **Pattern Parameters** - $ChildSwapping([X_1 : C_1], [X_2 : C_2] < X_2 : C_1 >, < X_1 : C_2 >$ $)$
- **Pattern Intent** - To swap the child components (X_1, X_2) from composite components (C_1, C_1).

Context and Forces

- **Constraints** - Represent the architecture model before, during and after changes.

1. *Preconditions* - X_1 is a child of composite component C_1, - X_2 is a child of composite component C_2
2. *Invariants* - C_1 and C_2 must be moved out of their parents C_1 and C_2.
3. *Postconditions* - X_2 is a child component of C_1, - X_1 is a child component of C_2.

- **Change Operators** - Enables the architecture change implementation.

1. *Rem(Component)* - Remove a Component X_1 from Component C_1.

2. *Add(Component)* - Add a Component X_1 into Component C_2.
3. *Rem(Component)* - Remove a Component X_2 from Component C_2.
4. *Add(Component)* - Add a Component X_2 into Component C_1.

∗ **Architecture Models** - Represents the affected architecture model.

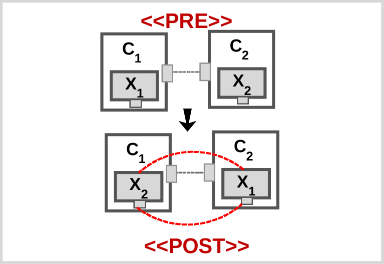


Figure D.12: Overview of the Child Swapping Pattern

A Template-based Specification of Component Mediation Pattern

1. Pattern Description

Pattern Classification - *CLS* <hasClassification = "Inclusion", classification ID = "1"> [Classifies]

Change Pattern - *PAT* <name = "Component Mediation", Intent = "Integration of a Mediator Component among existing Configuration(s)"> [composedOf | hasVariant | follows]

2. Pattern Context and Operators

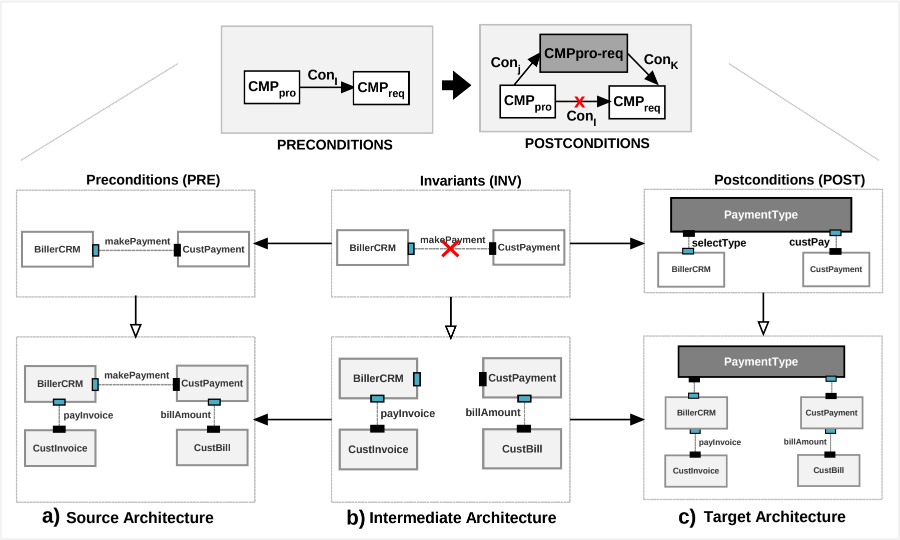
Change Operations - OPR <oprType = Add(); Rem(), compType = isComposite> [ConstrainedByEvolves]
- Add($CMP_{pro-req} \in CMP$);
- Add($CON_J \in CON, (CMP_{pro}, CMP_{pro-req}) \in CMP$);
- Add($CON_K \in CON, (CMP_{pro-req}, CMP_{req}) \in CMP$);
- Rem($CON_I \in CON, (CMP_{pro}, CMP_{req}) \in CMP$)

Pattern Constraints - CNS <PRE, INV, POST>
- **PRE** : $CON_I(CMP_{pro}, CMP_{req}) \in CON$
- **INV** : $\forall cmp \in CMP_{pro}, CMP_{pro-req}, CMP_{req} \exists por \in POR$
  $\forall con \in CON_I, CON_J, CON_K \exists ept \in EPT$
- **POST** : $(CON_J(CMP_{pro}, CMP_{pro-req}) \wedge CON_K(CMP_{pro-req}); CMP_{req})) \in CON$

3. Pattern Impacts on Architecture Model

ArchitectureModel - ARCH <PRE, INV, POST>
- **CMP** = $CMP_{pro}, CMP_{pro-req}, CMP_{req} \in CMP$
- **CON** = $CON_I, CON_J, CON_K \in CON$

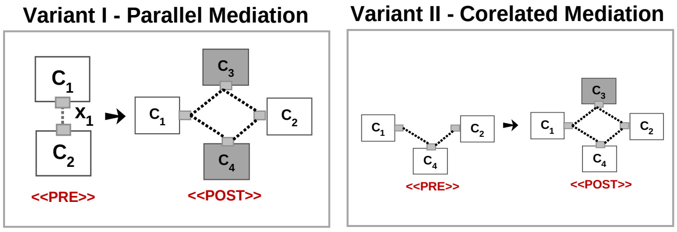
The pre-condition in the constraints specifies that provider and requester must be connected and the invariant(s) preserved. . The post-condition specify that provider and requester must now be connected through a mediator (where $CON_I \in CON_\phi$ is an orphaned connector that must be removed)
Pattern Instance: The generic specification above can be instantiated with concrete architecture elements. We integrate a new Payment Type service among directly connected services Pay Invoice and Transfer Money



Instance of Linear Inclusion Pattern for Component Integration(Partial Architecture View)

4. Pattern Variants

Variant - VAR< $PatternID_1, \ldots, PatternID_n$ >
It refers to the possible variants of the Linear Inclusion Pattern that are summarised as:
PatternID$_1$(Parallel Mediation) - refers to addition/removal of the architectural components that provide alternative/parallel functionality to an existing component, illustrated in Figure below. For example, to add a new component (D) that allows a redirection as (A, D, B) in addition to an existing connection (A, C, D)
PatternID$_n$(Corelated Mediation) - refers to adding/removing a set of functionally co-related architecture components into the existing architecture. For example, while applying observer pattern, addition/removal of an abstract observer requires addition/removal of a concrete observer in same change step to complete observer pattern application


Variant I - Parallel Mediation
Variant II - Corelated Mediation
C1
X1
C2
C3
C4
<<PRE>>
<<POST>>

Instance of Linear Inclusion Pattern for Component Integration(Partial Architecture View)

Table D.1: Example of Pattern 1 - Component Mediation Pattern.

Appendix E

Experimental Setup, Validity Threats and Questionnaire for Evaluation of the PatEvol Framework

Contents

E.1 Quality Sub-characteristics of ISO/IEC 9126 Model for Framework Evaluation

Quality Characteristic I - Functionality The evaluation of PatEvol functionality refers to the capability of the framework to provide functions/processes (architecture change mining and architecture change execution) which satisfies the needs and objectives of the solution framework when used under specified conditions. The underlying question we aim to investigate is:

Are the required functions of architecture change mining (for knowledge acquisition) and architecture change execution (for knowledge application) provided by the PatEvol framework?

We evaluate framework functionality based on the quality sub-characteristics with functional *suitability* and functional *accuracy*.

Sub-characteristic A - Suitability

It refers to the evaluating the capability of the framework to resolve the problems of recurring architecture evolution faced by the users/architects. More specifically, the framework must ensure a suitable mechanism to capture user inputs, process them in a correct and timely manner and produce the desired results. For example, during architecture change execution process the framework must ensure i) specifications of source architecture model, ii) enable pattern-driven architecture architectural transformation , and iii) represent the evolved architecture model.

Sub-characteristic B - Accuracy

It refers to evaluating the correctness and completeness of the framework in terms of producing accurate results. More specifically the correctness of framework implies that for each given input by user, the capability of the framework to produce an error-free and accurate output. For example, during architecture change mining process when the user specifies the change log graph along with parameters - minimum and maximum lengths of pattern candidates and pattern frequency threshold - the framework must ensure accuracy of pattern discovery.

E.1.1 Quality Characteristic II - Usability

The evaluation of PatEvol usability refers to the capability of the framework to be understood, learned and easily used by its user when used under specified conditions. The underlying question we aim to investigate is:

Is the PatEvol framework understood and of practical use to its users?

We evaluate framework **usability** based on the quality sub-characteristics *understandability* and *operatability*.

Sub-characteristic A - Understandability

It refers to evaluating the framework based on its understandability of the inputs, outputs and functionality. For example, during the architecture change mining process the user must be able to understand the role of the framework in terms of user inputs for pattern discovery. In addition, a clear understanding of the framework inputs and outputs enables the user a parameterised customisation of the pattern discovery process.

Sub-characteristic B - Operatability

refers to evaluating the framework based on its ease of operatability by its users in terms of providing inputs, viewing outputs and necessary intervention and supervision of its functionality, if required.

E.1.2 Quality Characteristic III - Efficiency

The evaluation of PatEvol efficiency refers to its capability to provide the required performance - processing time - in relation to the amount of resource utilisation under the stated conditions. The underlying question we aim to investigate is:

How efficient is the framework to enable architecture change mining and architecture change execution processes?

For example, in the architecture change mining process evaluating pattern discovery, algorithmic accuracy (correctness and completeness), and performance are the key factors to evaluate the efficiency. The accuracy of the algorithms is to identify all change patterns from architecture change logs. The evaluation must ensure the framework does not skip any available patterns. We focus on:

Sub-characteristic A - Time Behaviour

It refers to evaluating the time-efficiency of the framework. For example, in an architecture change mining process we are concerned with evaluating the computational complexity of pattern discovery algorithms.

Sub-characteristic B - Resource Utilisation

It refers to evaluating the utilisation of computational resources by the PatEvol Framework.

To complement such theoretical claims and the quality model we must derive a concrete evaluation strategy for an experimental evaluation of the framework and its results.

E.2 Experimental Setup for the Framework Evaluation

In this section, first, in Table E.1 we identify the evaluation methods for solutions (process and activities in PatEvol framework) that address different research challenges as identified in Chapter 1. Second, in Figure E.1 we present an activity-based view of setting-up and executing the individual activities in the evaluation process.

E.2.1 Identification of Evaluation Methods

Research challenges and proposed solutions are mentioned in Chapter 1 (Research Questions) that need to be evaluated based on the sub-quality characteristics from ISO/IEC 9126 - 1 product quality model - as summarised in Table E.1.

1. **Challenge I - Modelling Architecture Change Log Data** In order to enable change operation classification and pattern discovery from logs, a critical challenge lies with selection of an appropriate data structure to model change log data. It requires a careful selection of a data structure that enables a formal representation of log data along with efficient searching and retrieval of log data.

Research Problems and Solution		Evaluation Method			
Research Challenge	Solution	Sub-quality Characteristics	Evaluation Strategy		
(Research Questions)	(PatEvol Framework)	(ISO/IEC 9126 - 1)	Experiments	Feedback	ALMA
Challenge I	Modelling Log Graph	Efficiency, Suitability	✓	✓	✓
Challenge II	Pattern Discovery Algorithms	Accuracy, Efficiency	✓	✓	✓
Challenge III	Pattern Selection with QOC	Accuracy	✓	✓	
Challenge IV	Pattern-based Evolution	Efficiency, Reusability	✓		✓

Table E.1: Overview of Research Challenges, their Solutions and Evaluation Methods.

2. **Challenge II - Discovery of Architecture Change Patterns from Logs** Once log data is formalised as a graph (Challenge I), the challenges lies with an automated discovery of architecture change patterns from logs. The solution must ensure the discovery of exact as well as inexact instances of architecture change patterns.

3. **Challenge III - Pattern Selection form Language Collection** After pattern specification (Challenge III), the solution must support its user to select the most appropriate pattern(s) from language collection. Pattern selection requires a mapping of the problems of architecture evolution to select the applicable patterns as its solution.

4. **Challenge IV - Pattern-based Architecture Change Execution** After pattern selection (Challenge IV), the solution must support architecture evolution that is guided by change patterns. Pattern-based change execution requires patterns to guide source to target architecture transformation.

E.2.2 Activity I - Selected Case Studies for Architecture Evolution

In software engineering research, case study based approaches facilitate i) designing research process (Chapter 1 to Chapter 4), ii) conducting the research (Chapter 5 to Chapter 8) and evaluating research results (in this Chapter 9) by ensuring that data is collected and analysed in a systematic and scenario-driven environment [Flyvbjerg 2006]. We mentioned the the case studies in Chapter 2 - research background - and provide details about individual case studies in **Appendix B**.

E.2.3 Activity II - Collection of Change Log Data for Evaluation

The evaluation data is gathered by capturing architectural changes on EBPP and 3-in-1 Phone System case studies (Appendix B) as presented in Table E.2. Table E.2 presents all the architectural changes as number of change operations on architecture elements[1]. For example, the operation Add an architecture element that is of type Component with an occurrence frequency 212 represents addition of 212 architectural components recorded in change log. In Table E.2, we only provide an overview of addition removal and modification of configurations among a set of components (containing ports) and their connectors (containing endpoints).

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

Table E.2: Total Change Operations on Architecture Model Recorded from Change Log.

[1] The corresponding change log file recording change operations on architectural elements from Table E.2 is provided at: http://ahmadaakash.wix.com/aakash/LogFile.txt

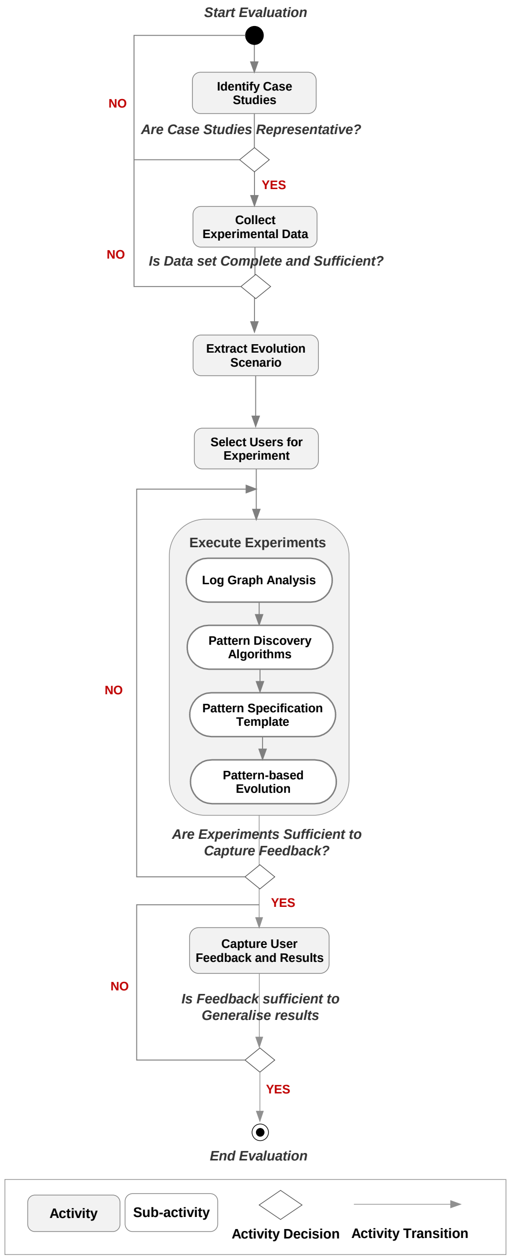

Start Evaluation
Identify Case Studies
Are Case Studies Representative?
NO
YES
Collect Experimental Data
Is Data set Complete and Sufficient?
NO
Extract Evolution Scenario
Select Users for Experiment
Execute Experiments
Log Graph Analysis
Pattern Discovery Algorithms
Pattern Specification Template
Pattern-based Evolution
NO
Are Experiments Sufficient to Capture Feedback?
YES
Capture User Feedback and Results
NO
Is Feedback sufficient to Generalise results
YES
End Evaluation
Activity
Sub-activity
Activity Decision
Activity Transition

Figure E.1: Activity-based Representation of Experimental Setup.

E.2.4 Activity III - ALMA-based Selection and Analysis of Architecture Evolution Scenarios

The ALMA method is used to select and analyse the architecture evolution scenarios to evaluate the architecture change mining and architecture change execution processes. We select these scenarios from architecture case studies (Activity II), as illustrated in Figure E.1. More specifically, we select two evolution scenarios from EBPP case study to analyse the accuracy and efficiency of pattern discovery algorithms in comparison to a manual pattern discovery presented in Chapter 7. In addition, we select four evolution scenarios to analyse the pattern-driven architecture evolution that updates a peer-to-peer system to client server architecture presented in Chapter 9 and Appendix D.

The ALMA method enables a scenario-driven approach for predicting (the efforts of) architectural maintenance and evolution and analysing the evolved architecture model as a 5-step process. Please note that, while following ALMA we do not aim to predict maintenance efforts, instead we only focus on *elicitation*, *evaluation* and *interpretation* of architecture evolution scenarios - Step III, IV, V as listed below. The ALMA method consists of the following five steps:

- **Step I** Set the goal for architecture-level evolution analysis.
- **Step II** Specify architecture description to provide a representation of the architecture model (before and after architectural evolution).
- **Step III** Elicit architecture evolution scenarios from case studies.
- **Step IV** Evaluate architecture scenarios to determine their effects on architecture descriptions.
- **Step V** Interpret results of evolution scenarios to draw conclusions from analysis.

E.2.5 Step IV - Selection of Participants for Experimental Feedback

To select the participants for evaluations, we also followed the key informant methodology - a qualitative method - to seek the participants'/interviewees' expert opinion or knowledge to develop or evaluate a solution [Gallivan 2001]. By following the guidelines of key information methodology we focused on, i) Relevance of the participants' expertise to the type of evaluation (i.e; software design and evolution process), ii) Design of evaluation activities (detailed in Chapter 9 and in this appendix) to seek the feedback for evaluation.

A participant-based evaluation of the PatEvol framework allows us to capture the feedback from participants in terms of evaluating the functional suitability and usability of the framework. We selected a total of five unique participants to evaluate the framework as presented in Table E.3. In Table E.3, each participant is assigned a unique id to maintain details of individual feedback. For example, in Table E.3 P1 who is an academic researcher in software architecture evolution to evaluate the framework is identified as P1.

In terms of the professional affiliation of the participants, we had a total of 3 participants from academia working as Ph.D researchers and two participants working in software industry[2] The academic participants are all working as software engineering (SE) researchers, while the industrial participants are from software design and development and software testing background. The total combined experience of participants in software engineering is 11 years with an average experience slightly more than 2 years per participant. More specifically, the participants have a combined experience of 8 years with software architecture (SA) - on average more than 1.5 years of an individual's experience. The participants had experience with architectural design, maintenance and validation. The professional or research expertise of researchers is architecture evolution (2 participants) and source code re-factoring (1 participant), while industrial professional have expertise in Java based development and UML 2.0 for software design.

Participant ID	**Professional Affiliation**	**Professional Role**	**Experience with SE (years)**	**Experience wwith SA (years)**	**Professional/Research Expertise**
P1	Academic Researcher	Research in SE	2	2	Architecture Evolution
P2	Academic Researcher	Research in SE	2	1	Architecture Evolution
P3	Academic Researcher	Research in SE	3	3	Code Refactoring
P4	Industrial Professional	Software Development	3	1	UML 2.0, JAVA
P5	Industrial Professional	Software Testing	1	1	UML 2.0, JSystem

Table E.3: Professional Affiliations, Role, Experience and Expertise of Participants for Feedback.

[2] Please note that in this evaluation the industrial professional do not represent their company or professional institute. Both the professional took part in the evaluation in their individual capacity. We aim to seek the feedback for evaluation both from academic researchers as well as from industrial professionals.

E.3 Evolution Scenarios for Pattern Discovery

Architecture Evolution Scenarios are identified using ALMA method (in Section E.2) and briefly present our goal for scenario analysis, architecture descriptions in the evolution scenarios, elicitation and evaluation of evolution scenarios before interpreting the results.

Goal(s) of the Analysis

The primary goal of the analysis is to analyse the accuracy and efficiency of the pattern discovery algorithms. An effective measure of accuracy is to investigate the precision and recall factor for discovery algorithms (discussed in Chapter 7). In addition, we also compare the efficiency and accuracy of the proposed pattern discovery algorithms by comparing the results with a manual discovery (i.e; automated vs manual). Such an analysis helps us to justify, if the discovery algorithms are more efficient and accurate than any efforts of manual discovery by interpreting the results of evolution scenarios.

Architecture Descriptions

In the evolution scenarios, architecture elements represent the parametrisation of change operations - i.e; change operations are applied to architecture elements. Therefore, architecture descriptions are provided using a graph based notation [Brandes 2002a] (graphml file) - a sample provided in Listing E.1. Listing E.1 represents the source architecture model before evolution (PRE as preconditions), changes on source architecture model (OPR as change operations) and target architecture model after evolution (POST as postconditions). The file contains a total 8 evolution scenarios, where some of the scenarios are recurrent (pre/post-conditions and operations remain same but architecture elements are different) representing the patterns in the file.

Listing E.1: A Sample of Log File for Manual Discovery of Change Patterns

```
 .......
<PRE>
  <data key="ArchElement"> BillerCRM </data>
  <data key="hasParam"> </data>
  <data key="hasType"> CMP </data>
  <data key="ArchElement"> CustPayment </data>
  <data key="hasParam"> </data>
  <data key="hasType"> CMP </data>
  <data key="ArchElement"> makePayment </data>
  <data key="hasParam"> BillerCRM, CustPayment </data>
  <data key="hasType"> CON </data>
</PRE>
<OPR>
  <data key="opr"> ADD </data>
  <data key="hasParam1"> PaymentType </data>
  <data key="Param1Type"> CMP </data>
  <data key="hasParam2"> </data>
  <data key="Param2Type"> </data>
  <data key="opr"> ADD </data>
  <data key="hasParam1"> selectType </data>
  <data key="Param1Type"> CON </data>
  <data key="hasParam2"> BillerCRM, PaymentType </data>
  <data key="Param2Type"> CMP </data>
  <data key="opr"> ADD </data>
  <data key="hasParam1"> custPay </data>
  <data key="Param1Type"> CON </data>
  <data key="hasParam2"> PaymentType, CustPayment </data>
  <data key="Param2Type"> CMP </data>
  <data key="opr"> REM </data>
 <data key="hasParam1"> makePayment </data>
  <data key="Param1Type"> CON </data>
  <data key="hasParam2"> BillerCRM, CustPayment </data>
  <data key="Param2Type"> CMP </data>
</OPR>

</POST>
  <data key="ArchElement"> BillerCRM </data>
  <data key="hasParam"> </data>
  <data key="hasType"> CMP </data>
  <data key="ArchElement"> CustPayment </data>
  <data key="hasParam"> </data>
  <data key="hasType"> CMP </data>
  <data key="ArchElement"> PaymentType </data>
  <data key="hasParam"> </data>
  <data key="hasType"> CMP </data>
  <data key="ArchElement"> selectType </data>
  <data key="hasParam"> BillerCRM, PaymentType </data>
  <data key="hasType"> CON </data>
  <data key="ArchElement"> custPay </data>
  <data key="hasParam"> PaymentType, CustPayment </data>
  <data key="hasType"> CON </data>
</POST>
 .......
```

Elicitation of Evolution Scenarios

We select two evolution scenarios in Figure E.2 and Figure E.3 that are presented to the participant to discover them in the sample file in Listing E.1. These evolution scenarios are the potential patterns in the log file depending on their frequency in the file. For example, *we asked the participants to discover these scenarios in the log file and report them as patterns if their frequency is 2 or more*. These evolution scenarios are presented to the participants in the form of Figure E.2 and Figure E.3. We provide the generic names for components and connectors and it is up to the participant to discover the pattern (if exists) with concrete name of architectural elements in sample log file. For example, in Figure E.2 the preconditions of a scenario represent that two components (A, B) are interconnected using a connector (X1).

- **Scenario I for Pattern Discovery** This scenario is selected based on the running example used in previous chapters (Chapter 7, Chapter 8). It represents the interposition of a mediator component PaymentType that facilitates the selection of a payment type mechanism among the directly connected components BillerCRM and CustPayment as illustrated in Figure E.2.

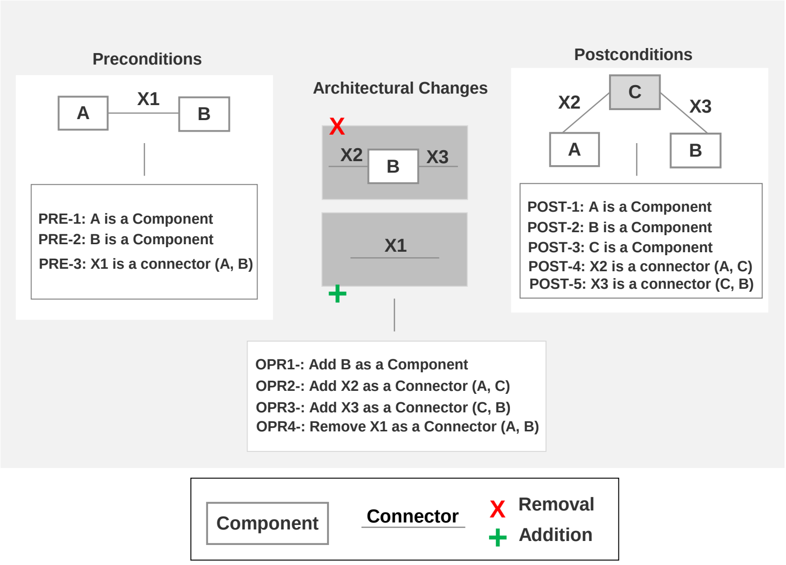

Preconditions
A
X1
B
PRE-1: A is a Component
PRE-2: B is a Component
PRE-3: X1 is a connector (A, B)
Architectural Changes
X
X2
B
X3
X1
+
OPR1-: Add B as a Component
OPR2-: Add X2 as a Connector (A, C)
OPR3-: Add X3 as a Connector (C, B)
OPR4-: Remove X1 as a Connector (A, B)
Postconditions
X2
C
X3
A
B
POST-1: A is a Component
POST-2: B is a Component
POST-3: C is a Component
POST-4: X2 is a connector (A, C)
POST-5: X3 is a connector (C, B)
Component
Connector
X Removal
+ Addition

Figure E.2: Overview of Scenario I as Presented to participants for Discovery in Sample Log File.

- **Scenario II for Pattern Discovery** represents the replacement of an existing component B with a new component C as presented in Figure E.3.

After presenting the scenario I (Figure E.2) and Scenario II (Figure E.3), we ask the participants to identify them in the sample log file (Listing E.1) with a specific occurrence frequency.

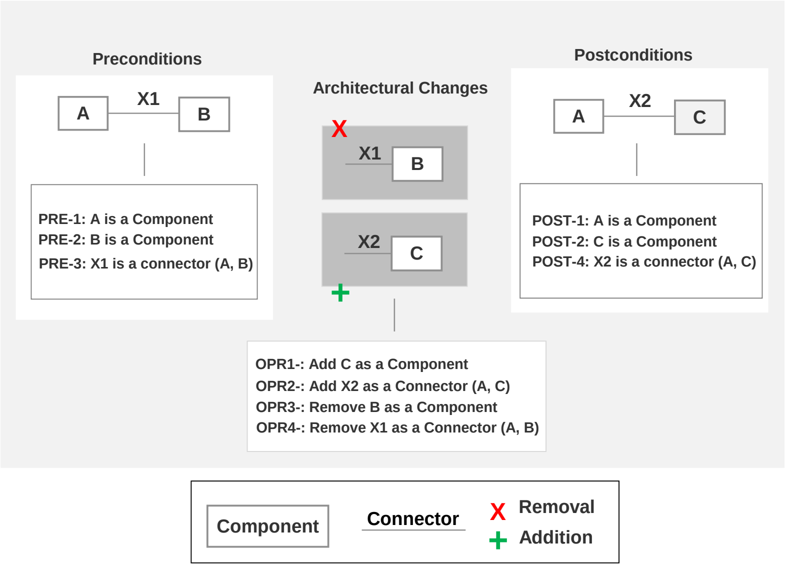


Figure E.3: Overview of Scenario II as Presented to Participants for Discovery in Sample Log File.

E.3.1 User Interfaces for Architecture Evolution Prototype

In this section, we present the user interfaces for pattern-driven architecture evolution by utilising the example from Figure 8.8 (cf. Section 8.6.2).

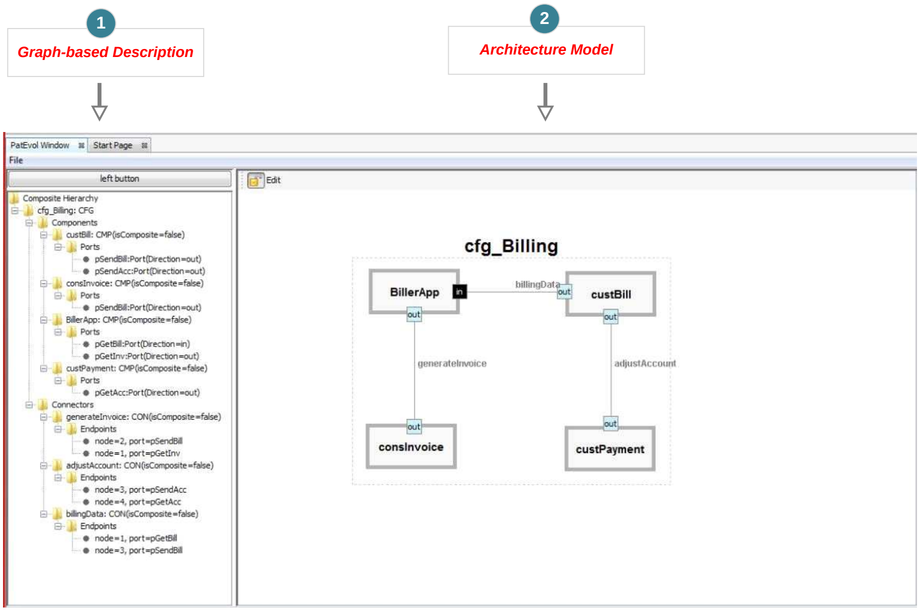
- **Interface to Import the Architecture Descriptions -** in Figure E.4, the interface imports the a) graph-description for source architecture model (left-hand side) and also b) visualises the configurations, components (their ports) and connectors (right-hand side).

- **Interface for Change Specification -** to specify changes on architecture model, the interface in Figure E.5 allows the user to specify the a) change rule to add or remove the desired architecture elements and b) the constraints as pre-conditions and post-conditions on the source architecture model.

- **Interface for Pattern Selection -** once a change rule is specified the prototype provides the most appropriate pattern to address the given evolution scenario in Figure

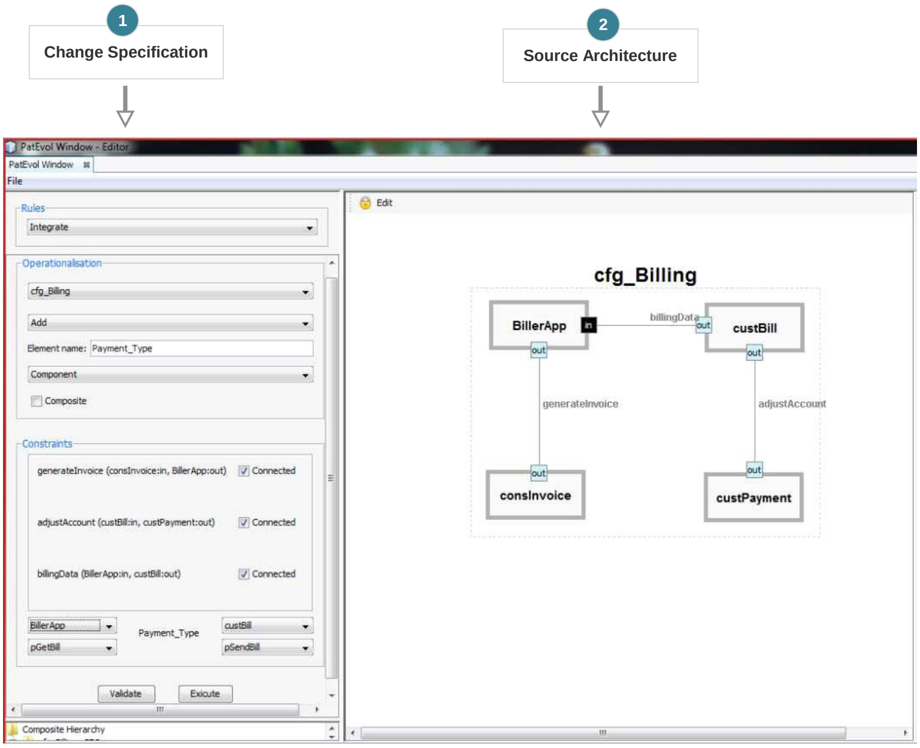
E.6. The interface presents the pattern description (left-hand side) including the *name*, *intent* and *change operationalisation*. In addition, the interface also provide an overview of the impact of change pattern on architecture model even before a pattern is applied. For example, in Figure E.6 the pattern impact shows interposition of a PaymentType component among directly connected components BillerApp and CustBill inside cfg-Billing configuration.

– **Interface for Change Execution -** once a pattern is applied, architectural changes are executed by abstracting the operation level details. The evolved architecture model is presented in Figure E.7. Graph-based description of the evolved architecture model is presented on the left-side, while a visualisation of the evolved components and connectors on the right.



```
Composite Hierarchy
cfg_Billing: CFG
  Components
    custBill: CMP(isComposite=false)
      Ports
        pSendBill:Port(Direction=out)
        pSendAcc:Port(Direction=out)
    consInvoice: CMP(isComposite=false)
      Ports
        pSendBill:Port(Direction=out)
    BillerApp: CMP(isComposite=false)
      Ports
        pGetBill:Port(Direction=in)
        pGetInv:Port(Direction=out)
    custPayment: CMP(isComposite=false)
      Ports
        pGetAcc:Port(Direction=out)
  Connectors
    generateInvoice: CON(isComposite=false)
      Endpoints
        node=2, port=pSendBill
        node=1, port=pGetInv
    adjustAccount: CON(isComposite=false)
      Endpoints
        node=3, port=pSendAcc
        node=4, port=pGetAcc
    billingData: CON(isComposite=false)
      Endpoints
        node=1, port=pGetBill
        node=3, port=pSendBill
```

Figure E.4: User Interface to Import the Source Architecture Model.



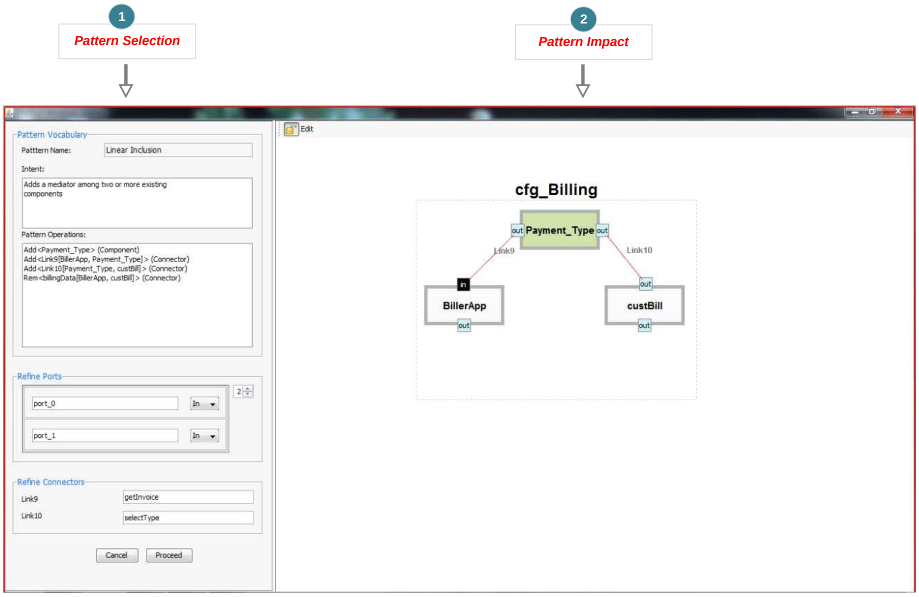
Rules: Integrate

Operationalisation:
cfg_Billing
Add
Element name: Payment_Type
Component
□ Composite

Constraints:
generateInvoice (consInvoice:in, BillerApp:out) ☑ Connected
adjustAccount (custBill:in, custPayment:out) ☑ Connected
billingData (BillerApp:in, custBill:out) ☑ Connected

BillerApp — Payment_Type — custBill
pGetBill — pSendBill

Figure E.5: User Interface to Specify Architectural Changes.

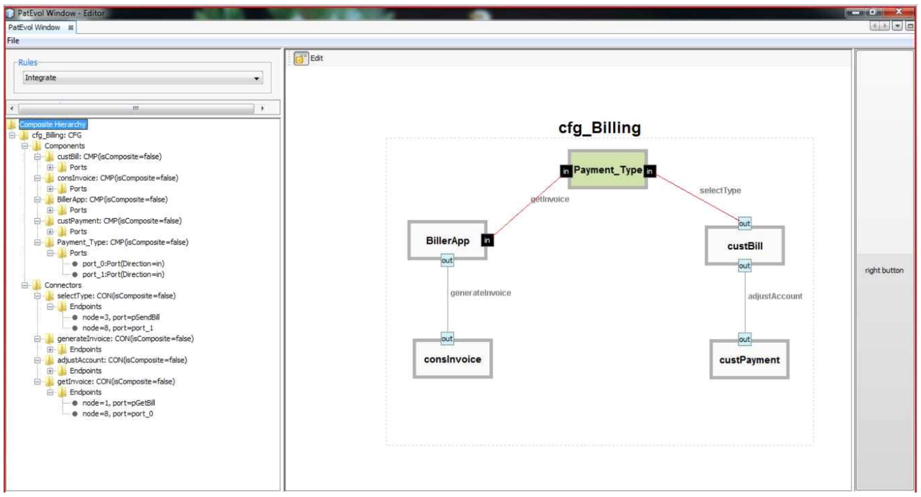


Pattern Vocabulary
Patttern Name: Linear Inclusion
Intent:
Adds a mediator among two or more existing components
Pattern Operations:
Add <Payment_Type> (Component)
Add <Link9[BillerApp, Payment_Type]> (Connector)
Add <Link10[Payment_Type, custBill]> (Connector)
Rem <billingData[BillerApp, custBill]> (Connector)

Refine Ports
port_0 In
port_1 In
2

Refine Connectors
Link9 getInvoice
Link10 selectType

Figure E.6: User Interface to Select Change Patterns.



```
Composite Hierarchy
  cfg_Billing: CFG
    Components
      custBill: CMP(isComposite=false)
        Ports
      consInvoice: CMP(isComposite=false)
        Ports
      BillerApp: CMP(isComposite=false)
        Ports
      custPayment: CMP(isComposite=false)
        Ports
      Payment_Type: CMP(isComposite=false)
        Ports
          port_0:Port(Direction=in)
          port_1:Port(Direction=in)
    Connectors
      selectType: CON(isComposite=false)
        Endpoints
          node=3, port=pSendBill
          node=8, port=port_1
      generateInvoice: CON(isComposite=false)
        Endpoints
      adjustAccount: CON(isComposite=false)
        Endpoints
      getInvoice: CON(isComposite=false)
        Endpoints
          node=1, port=pGetBill
          node=8, port=port_0
```

Figure E.7: User Interface for Description of Evolved Architecture Model.

E.4 Questionnaire for Participant's Feedback

lero THE IRISH SOFTWARE ENGINEERING RESEARCH CENTRE

DCU

Software and System Engineering Research Group

School of Computing, Dublin City University.

General Instructions

– Please indicate the option that satisfies your opinion the most with a tick (✓) in the specific field ([✓]) (you are allowed to mark more than one options, where you feel appropriate).

- Answers to the provided questions can be written or recorded. Should you feel you need to record your answers please ask the coordinator before starting the questionnaire.
- Please ask the coordinator for explanations or clarifications any-time when answering the questionnaire. Feel free to indicate either if you are unable understand something or you want to point any issue related to the questionnaire.

1. Participant’s Profile

Name	
Affiliation	
Professional Role	
On a scale from 1 to 5, where 1 represents *"Expert"* and 5 represent *"No knowledge/expertise"*, please indicate your expertise about software architecture or software design. 1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ]	
On a scale from 1 to 5, where 1 represents *"Expert"* and 5 represent *"No knowledge/expertise"*, please indicate your expertise about any of the following Maintenance OR evolution OR adaptation of Software OR Software Architecture OR Software Design. 1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ]	
On a scale from 1 to 5, where 1 is *"Expert"* and 5 is *"No knowledge/expertise"*, please refer to your knowledge about any of the following Design Patterns OR Architecture Styles 1 [ ] 2 [ ] 3 [ ] 4 [ ] 5 [ ]	

2. Sections of the Questionnaire

The questionnaire captures the participant feedback based on the experiments for the analysis and evaluation purposes.

- **Part I** Feedback for Suitability of Change Log Graph for Architecture Change Representation.

- **Part II** Feedback for Accuracy and Efficiency of Pattern Discovery Algorithms.

- **Part III** Feedback for Efficiency and Re usability of Pattern-based Evolution.

E.5 Part I - Suitability of Change Log Graph for Architecture Change Representation

This section aims to capture the participants' feedback for evaluating the *Suitability* of Log Graph. Suitability is a sub-characteristics of Functionality in ISO 9126 - 1 model. Efficiency of log-graph is evaluated based on experimental analysis for time required to retrieval data from log graph.

E.5.1 Instructions for the Participants

- *Step I* Analyse the sample change log file that is provided with the questionnaire.
- *Step II* Analyse the sample change log graph that is provided with the questionnaire
- *Step III* Identify a change operation (both in the log file and log graph) that enables addition of an architectural component PaymentType
- *Step IV* Convert at-least 3 change operations from change log file into a change log graph.
- *Step V* Addition of change operations in the change log file and repeat Step IV.
- *Step VI* Add a new configuration, a component containing a port in the configuration as an entry in the change log file.
- *Step VII* Add a new configuration, a component containing a port in the configuration as an entry in the change log graph file.

E.5.2 Questions to the Participants

- *Question I* Which of the two provides a Suitable representation of change operationalisation on architecture elements?

  Log File [ ] **Log Graph** [ ] **Not Sure** [ ]

  Comments: ____

- *Question II* Which of the two provides an easy interpret of the intent of architecture change operations?

Log File [ ]	**Log Graph** [ ]	**Not Sure** [ ]

Comments: ____

- *Question III* Visualisation of changes on architecture elements can be better achieved with?

  Log File [ ]	**Log Graph** [ ]	**Not Sure** [ ]

  Comments: ____

- *Question IV* It is easy to search and retrieve the log data from?

  Log File [ ]	**Log Graph** [ ]	**Not Sure** [ ]

  Comments: ____

- *Question V* It is Easy to record change operations in?

  Log File [ ]	**Log Graph** [ ]	**Not Sure** [ ]

  Comments: ____

E.6 Part II - Accuracy and Efficiency of Log Graph

This section aims to capture the participants' feedback for evaluating the *Accuracy* and *Efficiency* of Log Graph. Accuracy and efficiency are sub-characteristics of ISO 9126 - 1 quality model.

E.6.1 Instructions for the Participants'

- *Step I* Analyse the architecture evolution scenarios provided with the questionnaire.
- *Step II* Analyse the change log graph provided with the questionnaire.
- *Step III* Identify the occurrence of evolution scenarios (Step I) in the log graph file (Step II).

E.6.2 Questions to the Participants

- *Question I* Please write down the number of pattern instances discovered from log graph.

Number of Discovered Pattern(s): ____________________

- *Question II* Please write down the time taken to discover pattern instances from log graph.

  Time taken in Second(s): ____________________

- *Question III* Do you notice any overlap of change operations on discovered pattern instances?

  YES [ ] **NO** [ ] **Not Sure** [ ]

  Comments: ____________________

E.6.3 Filled By Coordinator

- *Result I* What type of pattern instances are discovered by the participant?

  Exact Instances [ ] **Inexact Instances** [ ] **Both** [ ]

  Comments: ____________________

- *Result II* What is the pattern discovery precision by the participant?

  1.0 [ ] $\leq$ **1.0** $\geq$ **0.5** [ ] $\leq$ **0.5** [ ]

  Comments: ____________________

- *Result III* Is candidate identification required?

  YES [ ] **NO** [ ] **Not Sure** [ ]

  Comments: ____________________

E.7 Efficiency and Reusability of Architecture Evolution

This section aims to capture the participants′ feedback for evaluating the efficiency and reusability of pattern-based architecture evolution. Accuracy and efficiency are sub-characteristics of ISO 9126 - 1 quality model.

E.7.1 Instructions for the Participants

- *Step I* Analyse the architecture evolution scenarios.
- *Step II* Specify the architectural changes to support the scenario.

E.7.2 Questions to the Participants

- *Question I* Does the Pattern-based evolution support reuse of architectural changes?

  YES [ ] **NO** [ ] **Not Sure** [ ]

  Comments: ____

- *Question II* Is the process understandable?

  YES [ ] **NO** [ ] **Not Sure** [ ]

  Comments: ____

- *Question III* Is Pattern-based architecture evolution process more efficient?

  YES [ ] **NO** [ ] **Not Sure** [ ]

  Comments: ____

End of Questionnaire

Thank you for your participation!

Comments:

____