COEVOLUTION FRAMEWORK TO SUPPORT OBJECT-ORIENTED MODEL CHANGES USING COLOURED PETRI NET PATTERNS

BASSAM ATIEH M.RAJABI

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

COEVOLUTION FRAMEWORK TO SUPPORT OBJECT-ORIENTED MODEL CHANGES USING COLOURED PETRI NET PATTERNS

BASSAM ATIEH M.RAJABI

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND
INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

UNIVERSITI MALAYA
ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: **Bassam Atieh M.Rajabi**

Registration/Matric No: **WHA080009**

Name of Degree: **DOCTOR OF PHILOSOPHY**

COEVOLUTION FRAMEWORK TO SUPPORT OBJECT-ORIENTED MODEL CHANGES USING COLOURED PETRI NET PATTERNS

Field of Study: Software Components Change Management

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date

1/2/2017

Subscribed and solemnly declared before,

Witness's Signature Date

Name: Prof. Lee Sai Peck

27/9/2017

Designation:

COEVOLUTION FRAMEWORK TO SUPPORT OBJECT-ORIENTED MODEL CHANGES USING COLOURED PETRI NET PATTERNS

ABSTRACT

An effective change management technique is essential to keep track of changes and to ensure that software projects are implemented in the most effective way. One of the crucial challenges in software change management is to maintain coevolution among software system artifacts. Object-Oriented (OO) software modelling is widely adopted in software analysis and design. OO diagrams are divided into different perspectives in modelling a problem domain. Preserving coevolution among these diagrams is very crucial so that they can be updated continuously to reflect software changes. Decades of research efforts have produced a wide spectrum of approaches in checking coevolution among OO diagrams. These approaches can be classified into direct, transformational, formal semantics, or knowledge representation approaches. Formal methods such as Coloured Petri Nets (CPNs) are widely used in detecting and handling coevolution between software artifacts. Although ample progress has been made, it still remains much work to be done in further improving the effectiveness and accuracy of the state-of-the-art coevolution techniques in managing changes in OO diagrams using formal languages. In this research, a coevolution framework for supporting coevolution among OO diagrams is proposed to trace the diagrams' inconsistencies and to determine the change impact incrementally after updating diagrams elements. A set of 84 coevolution patterns is proposed to detect and resolve UML diagrams' coevolution, inconsistencies, change history, and change impact. Coevolution patterns are applied on UML class, object, activity, statechart, and sequence diagrams to cover the different perspectives of UML diagrams. The change impact and traceability analysis is performed with the help of templates. A total of 45 templates are proposed to define information about the types of change, change impact, diagrams dependency, and rules to maintain the diagrams' consistency. As part of the proposed framework, a new structure called Object Oriented Coloured Petri Nets (OOCPNs) for the mutual integration of UML and CPNs modelling languages is proposed to support coevolution between UML diagrams. The proposed structure combines the advantages offered by CPNs formal language and the structured capabilities offered by UML diagrams to solve the inconsistencies between UML diagrams by integrating a set of consistency and integrity rules in the transformation process of UML diagrams into CPNs model. As such, this research also provides transformation rules for the diagrams provided in UML 2.3. The proposed OOCPNs structure enhances the diagrams' change support through building a consistent OOCPNs model at the design time, and then applying the changes on the OOCPNs models. This will provide OOCPNs model automatic coevolution and consistency check. Additionally, the modularity in the hierarchical structure of the proposed framework reduces interdependencies between the model components, and facilitates easy maintenance and updates without impacting the entire model. The researcher uses CPNs as a formal language of modelling case study models for the proposed framework and CPNs Tools as the software that creates, simulates, and validates the models. CPNs tools simulation and monitoring toolboxes are used to validate the proposed coevolution framework models and to monitor and collect data about the proposed framework quantitative results.

keywords: Coevolution , Patterns , UML, Coloured Petri Net

RANGKA KERGA EVOLUSI-BERSAMA UNTUK MENYOKONG PERUBAHAN MODEL BERORIENTTASIKAN-OBJECT DENGAN MENGGUNAKAN CORAK PETRI NET BEWARNA

ABSTRAK

Teknik pengurusan perubahan yang berkesan adalah penting untuk ikut laluan perubahan dan memastikan bahawa projek perisian dilaksanakan dengan cara yang paling berkesan. Salah satu cabaran yang penting dalam pengurusan perubahan perisian adalah untuk mengekalkan coevolusi antara artifak sistem. Pemodelan perisian berorientasikan objek (OO) diterima dan dipakai secara meluas dalam analisis dan reka bentuk perisian. Gambar rajah OO dibahagikan kepada beberapa perspektif yang berbeza dalam pemodelan domain masalah. Memelihara coevolusi antara gambar rajah tersebut adalah amat penting supaya ia boleh dikemaskini secara berterusan untuk mencerminkan perubahan perisian. Usaha penyelidikan yang berdekad telah menghasilkan pelbagai pendekatan dalam memeriksa coevolusi antara gambar rajah OO. Pendekatan tersebut boleh dikelaskan kepada pendekatan langsung, transformasi, semantik formal, atau perwakilan ilmu. Kaedah formal seperti Petri Nets Berwarna (CPN) digunakan secara meluas dalam mengesan dan pengendalian coevolusi antara artifak perisian. Walaupun kemajuan yang mencukupi telah diusahakan, masih kekal lagi banyak kerja yang perlu dilakukan dalam meningkatkan lagi keberkesanan dan ketepatan teknik coevolusi dalam menguruskan perubahan gambar rajah OO menggunakan bahasa formal. Dalam kajian ini, satu rangka kerja coevolusi untuk menyokong coevolusi antara gambar rajah OO adalah dicadangkan untuk mengesan ketidakselarasan gambar rajah dan untuk menentukan kesan perubahan secara berperingkat selepas pengemaskinian suatu rajah elemen. Satu set yang mengandungi 84 corak coevolusi adalah dicadangkan untuk mengesan dan menyelesaikan masalah coevolusi gambar rajah UML dari segi ketidakselarasan dan kesan perubahan mengubah sejarah. Corak coevolusi digunakan pada gambar rajah UML termasuk kelas, objek, aktiviti, statechart, dan turutan untuk menampung perspektif yang berbeza daripada gambar rajah UML. Kesan perubahan dan analisasi susur galur dilakukan dengan bantuan template. Sebanyak 45 template telah dicadangkan kepada menetapkan maklumat mengenai jenis-jenis perubahan yang disokong, kesan perubahan, kebergantungan antara gambar rajah, dan kaedah-kaedah untuk mengekalkan konsisten gambar rajah. Sebagai sebahagian daripada rangka kerja yang dicadangkan, struktur baru yang dikenali sebagai Petri Nets Berwarna Berorientasikan Objek (OOCPNs) untuk integrasi bersama bahasa pemodelan UML dan CPNs adalah dicadangkan untuk menyokong coevolusi antara gambar rajah UML. Struktur yang dicadangkan menggabungkan kelebihan yang ditawarkan oleh bahasa formal CPNs dan keupayaan berstruktur yang ditawarkan oleh gambar rajah UML untuk menyelesaikan percanggahan antara gambar rajah UML. Satu set yang mengandungi 78 peraturan konsisten dan integriti disepadukan dalam proses transformasi gambar rajah UML untuk menghasilkan model CPNs. Oleh itu, kajian ini juga menyediakan peraturan transformasi bagi gambar rajah UML 2.3. Struktur OOCPNs yang dicadangan meningkatkan sokongan perubahan gambar rajah melalui pembinaan model yang konsisten dipanggil model OOCPNs pada masa reka bentuk dan kemudian memakai perubahan tersebut pada model OOCPNs. Ini akan memberikan model OOCPNs mempunyai keupayaan coevolusi dan penyemakan konsistent secara automatik. Selain itu, kebermodulan dalam struktur hierarki rangka kerja yang dicadangkan mengurangkan kebergantungan antara komponen model dan memudahkan penyelenggaraan mudah dan kemas kini tanpa menjejaskan keseluruhan model tersebut. Pengkaji menggunakan CPNs sebagai bahasa rasmi model kajian kes untuk rangka kerja yang dicadangkan dan alat CPNs sebagai perisian yang mencipta, menyerupai, dan mengesahkan model. Alat simulasi CPNs dan peralatan pemantauan digunakan untuk mengesahkan model rangka kerja coevolusi yang dicadangkan, dan untuk memantau dan mengumpul data mengenai keputusan kuantitatif rangka kerja yang dicadangkan.

Kata Kunci: Corak Coevolusi, UML, Corak Petri Net

ACKNOWLEDGEMENTS

First and foremost, Alhamdulillah, praise to Allah the Almighty who has bestowed upon me and given me good health and time to go through this challenging period of my life.

I would like to express my deepest appreciation to my supervisor, Professor Lee Sai Peck who had the foresight to accept me as her doctoral student, believed in my abilities, kept advising me to improve my knowledge and writing, and supported me at crucial moments of my study at the University of Malaya. Her patience, kindness, understanding, encouragement, careful reading of the manuscript and especially her constructive criticisms and suggestions made the process of conducting this research run smoothly. I have benefited from her experience, advice and the intellectual freedom she afforded to me during the research period.

I would also like to thank the University of Malaya's staff (especially Faculty of Computer Science and Information Technology staff) for providing impeccable assistance and academic resources. Thanks to Prof. Colette Rolland for her notes on my contribution. I am very thankful to the Wajdi Foundation for Educational Development for the financial support and Wajdi University College of Technology for the study leave that has enabled me to pursue my academic dreams. Last but not least, my appreciation also goes to those who were directly or indirectly involved in this research. These include my mother and father, my wife, my sisters and brothers. Special thanks and apologies to my loving wife Asma. May Allah bless you all for your kindness, support, encouragement and willingness to help in completing this research.

To many others who have helped me in one way or other during my course of study but whom I may have inadvertently left out, please kindly excuse me.

This research is supported by PPP University Malaya fund no. PS077/2009A.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF TRACEABILITY ANALYSIS AND CHANGE IMPACT TEMPLATES

LIST OF COEVOLUTION PATTERNS

LIST OF SYMBOLS AND ABBREVIATIONS

General	
BP	Business Process
BPs	Business Processes
BPMN	Business Process Modelling Notation
BPDM	Business Process Definition Meta-model
EPC	Event Driven Process Chain
iEPCs	Integrated Event driven Process Chains
IT	Information Technology
rBPMN	Rule-based BPMN
WS-BPEL	Web Services Business Process Execution Language
MDE	Model Driven Engineering
Object-Oriented	
AD	Activity Diagram
CD	Class Diagram
CSD	Composite Structure Diagram
CoD	Component Diagram
CommD	Communication Diagram
DD	Deployment Diagram
IOD	Interaction Overview Diagram
OD	Object Diagram
OO	Object-Oriented
OOD	Object Oriented Design
PD	Package Diagram
SCD	Statechart Diagram

SD	Sequence Diagram
TD	Timing Diagram
UCD	Use Case Diagram
UML	Unified Modelling Language

Petri Nets

CPN	Coloured Petri Net
CPNs	Coloured Petri Nets
PN	Petri Net
PNs	Petri Nets

Object Oriented Petri Nets

HCPN	Hierarchical Coloured Petri Net
HPN	High Level PNs
LPN	Low Level PNs
OOPN	Object Oriented Petri Nets
OOCPNs	Object Oriented Coloured Petri Nets
OOMPNets	Object Oriented Petri Nets with Modularity
OPMs	Object PN Models
RONs	Reconfigurable Object Nets

Coevoloution Framework

$\sum$	Finite set of non-empty types, called colour sets
A	Set of directed arcs
alt	alternative
B	Behavioural diagram’s elements
C	UML diagrams’ Categories
CI	Change Impact
E_o, N_o	UML Diagrams Elements

D	CI Dependency
Fp	Finite set of fusion places
G	Guard function
GC	Global Change
I	Interaction diagram's elements
LC	Local Change
M_0	Initial (coloured) marking
N	Diagram Name
opt	optional
P	Finite set of places
par	parallel
Pg	Set of CPN pages
R	Finite set of consistency and integrity rules
ref	reference
S	Structural diagram's elements
SubT	Finite set of substitution transitions
T	Finite set of transitions
TA	Traceability Analysis
TR	Transformation Rule
CT:	Change Type
AffectedD	Affected Diagrams (Dependency)
SSG	State Space Graph

LIST OF APPENDICES

CHAPTER 1: INTRODUCTION

Software change is continuous and unavoidable due to rapidly changing requirements across software systems. It is the result of adding new requirements of functionality, fixing faults, or change requests (Lehnert & Riebisch, 2013). Software change management describes a software system's ability to easily accommodate future changes. It is a fundamental characteristic for making strategic decisions, increasing economic value of software, and managing changes in an orderly fashion (Breivold, Crnkovic, & Larsson, 2012). An effective change management will lead organisations to the path of success, and it is an essential activity in the software project life cycle to keep track of changes and to ensure that they are implemented in the most effective way (Saif, Razzaq, Rehman, Javed, & Ahmad, 2013). Software engineers continue to face challenges in designing adaptive and flexible software systems that can cope with dynamic change where requirements are constantly changing (Khalil & Dingel, 2013; Lehnert & Riebisch, 2013; Nurcan, 2008). Unmanaged change may lead to fault-prone software, thereby increasing the testing and maintenance costs (Jönsson, 2005).

One of the crucial challenges in software change management is to preserve the coevolution and consistency among software system artefacts (Langhammer, 2013; Liu, 2013; Puissant, Van Der Straeten, & Mens, 2013). Understanding the coevolution which represents the dependency between artefacts that frequently change together is important from the points of views of both practitioners and researchers (Jaafar, 2012). Coevolution involves both change impact analysis and change propagation between software artefacts or models, and hence, it is required to (Dubauskaite & Vasilecas, 2013; Etien & Salinesi, 2005; Puczynski, 2012; Puissant, et al., 2013):

- Check if the change in one of the artefacts ultimately affects the other artefacts and may cause some unexpected changes in them,

- Ensure that these changes are implemented in the most effective manner, and
- Maintain the consistency between artefacts.

For an efficient coevolution check, change impact analysis is an important step. A change impact analysis is the activity of analysing and determining the change effect, identifying the parts that require retesting, and maintaining the consistency among software artefacts (Abma, 2009; C.-Y. Chen, She, & Tang, 2007; Li, Sun, Leung, & Zhang, 2012; Redding, 2009). Identifying all components affected by the change is based on the traceability analysis which analyses the dependencies between and across software artefacts at all levels of the software process (Mohan, Xu, Cao, & Ramesh, 2008). Detecting and resolving the coevolution between software artefacts can be done through various techniques. Some of these techniques are analysing release histories or versions, source code, and software architecture level analysis (Breivold, et al., 2012).

There are different approaches proposed in the literature that use these techniques to manage changes in the software project life cycle including changes in software requirements, design models, and programming code. Many of these approaches are focused on the coevolution of software modelling, in particular, Object-Oriented (OO) software modelling, due to its wide adoption in software modelling and design. The use of OO diagrams in modelling a software system leads to a large number of interdependent diagrams. OO diagrams are divided into different categories or perspectives (e.g. structural, behavioural, and interaction as elaborated in (Barr and Pettis (2007), Sharaff (2013), and Rajabi and Lee (2014))); each category focuses on modelling a different perspective of a problem domain. One of the critical issues in providing a change management technique for OO diagrams is to preserve the coevolution among these diagrams so that they can be updated continuously to reflect software changes (Langhammer, 2013; Liu, 2013; Lucas, Molina, & Toval, 2009; Puissant, et al., 2013; Shinkawa, 2006).

Decades of research efforts have produced a wide spectrum of approaches and techniques in checking the coevolution and inconsistency among OO diagrams. These approaches can be classified into direct, transformational, formal semantics, or knowledge representation approaches (Sapna & Mohanty, 2007). Direct approaches use the constructs of OO and Object Constraints Language (OCL) (Briand, Labiche, & O'sullivan, 2003; Briand, Labiche, & Yue, 2009). Transformational approaches derive a common notation by transforming one model to another (García, Diaz, & Azanza, 2013; Protic, 2011). Formal approaches develop formal semantics for the OO diagrams (Shinkawa, 2006), while knowledge representation approaches use description logics as a representation language (Bolloju, Schneider, & Sugumaran, 2012). A hybrid approach is a combination between two or more different type of these approaches (Khalil & Dingel, 2013).

According to Lucas et al. (2009), 75% of the approaches and techniques used for detecting and handling the coevolution and inconsistencies problems are formal. The most common formal methods used are state transitions methods such as Petri Nets (PNs). Although ample progress has been made, there still remains much work to be done in further improving the effectiveness and the accuracy of the state-of-the-art coevolution techniques in managing changes in OO diagrams using formal languages.

In this research, a coevolution framework for supporting coevolution among OO diagrams is proposed to trace the diagrams' inconsistencies and to determine the effect of change in these diagrams after each change operation. The proposed framework is used to check the consistency, impact, and traceability incrementally after creating, deleting, or modifying a diagram or diagram element. Additionally, a change history between two versions created from the same diagram is addressed in this research.

Unified Modelling Language (UML) is the standard language for modelling OO software (Bennett, McRobb, & Farmer, 2010; OMG, 2004, 2010). The coevolution and inconsistencies between UML diagrams will be detected and resolved based on a set of proposed coevolution patterns within the proposed coevolution framework. The concept of pattern was introduced by Christopher Alexander (1979). Alexander defined a pattern as:

> *"a three-part rule, which expresses a relation between a certain context, a problem, and a solution"* (1979, p. 247).

Patterns characterize the methods or techniques that have been encountered in practice repeatedly (Nataliya Mulyar & van der Aalst, 2005). Design patterns in OO design capture frequently recurring sub-designs or groups of objects that collaborate to perform a certain task (Gamma , Helm, Johnson , & Vlissides, 1995; Gamma, Helm, Johnson, & Vlissides, 2001).

The researcher studied the state of the art patterns mainly patterns proposed by (Alexander, 1979) and Gamma's (Gamma , et al., 1995; Gamma, et al., 2001) and proposed a new set of patterns to support coevolution between UML diagrams including change impact and traceability analysis of changes on diagrams elements. The change impact and traceability analysis is performed with the help of templates for all types of change in UML diagram elements. These templates define information about the types of change supported for each diagram, information on change impact, dependency between diagrams, and rules to maintain the integrity and consistency between diagrams.

The proposed patterns are the basis of initiation for all update operations, and are used to detect any elements affected by the change in systems modelled using UML diagrams. In the scope of this research, change impact and traceability analysis templates are defined for most of the diagrams' elements provided in UML 2.3. Coevolution patterns are applied on class, object, activity, statechart, and sequence diagrams. These diagrams cover the three perspectives of UML diagrams (i.e. structural, behavioural, and interaction).

UML is a powerful means for describing the static and dynamic aspects of systems (Bennett, et al., 2010; Bruegge, 2010), but remains semi-formal and lacks techniques for model validation and verification (Bousse, 2012; Niepostyn, 2015). According to Lucas (2009), formal specifications and mathematical foundations such as Coloured Petri Nets (CPNs) are widely used in handling of inconsistency problems among models and to automatically validate and verify the model dynamic behaviour (Kurt Jensen & Kristensen, 2009; Kurt Jensen, Kristensen, & Wells, 2007; Lucas, et al., 2009).

Due to the advantages offered by formal languages, the integration between UML and formal languages is recommended to solve the inconsistencies between UML diagrams (Lucas, et al., 2009). The advantages from the integration of UML and CPNs are better representation of a system's complexity as well as ease in adapting, correcting, analysing, and reusing a model. Transformation rules are required to transform UML diagrams elements to CPNs. Approaches discussed in the literature on the transformation of UML diagrams to CPNs focus on the part of UML diagrams, in particular, the behavioural diagrams. Additionally, the consistency check is based on a set of rules applied on the Coloured Petri Nets (CPNs) model.

In this research, as part of the proposed coevolution framework, a new structure for the mutual integration of UML and CPNs modelling languages is proposed to support the coevolution between UML diagrams. In the proposed structure, consistency and integrity rules are part of the transformation process and integrated in the transformed CPNs model. As such, this research also provides transformation rules for the diagrams provided in UML 2.3. The consistency rules include a set of rules to check and maintain the consistency and integrity based on the relations between UML diagrams. CPNs as a language of modelling are used to model case study models for the proposed framework. Additionally, CPNs Tools are used as software to creates, simulates, and validates the proposed framework models which represent the proposed transformation rules, templates, and coevolution patterns.

1.1 Problem Statement

Software change is inevitable in software project lifecycle. When new changes are applied to software, they would be having some impacts and inconsistencies with other parts of the original software (Li, et al., 2012). Software engineering researchers have stated that change management is concerned with what changes have been made and the effect of changes (Tam, Greenberg, & Maurer, 2000). Nowadays, effective change management is essential for organisational development and survival in order to keep track of changes and to reduce risks and costs (Saif, et al., 2013; Sommervile, 2007; Sommerville, 2011). Change management has been recognized as “the most difficult, costly and labour-intensive activity in the software development life cycle” Li et al. (2012).

One of the main issues in software change management is to detect and resolve the coevolution among software artefacts to determine the change impact and change propagation (Kchaou, Bouassida, & Ben-Abdallah, 2016; Langhammer, 2013; Liu, 2013; Lucas, et al., 2009; Puissant, et al., 2013; Shinkawa, 2006). Detecting and resolving the coevolution among software models is of tremendous significance for the field of software design and development to assess the change consequences. Software models are highly dynamic and evolve from requirements through implementation (Ivkovic & Kontogiannis, 2004). It is important to investigate how to integrate software changes into software models (April & Abran, 2012; Mens et al., 2005b). A change management technique is required to support the criteria of flexibility, adaptability, and dynamic reaction to changes in software models.

OO modelling is widely used in software analysis and design. It describes a system by modelling different perspectives using its structural, behavioural, and interaction diagrams. One of the crucial issues in checking the coevolution among OO diagrams is to control the change and to keep these different views or perspectives consistent (Dubauskaite & Vasilecas, 2013; Puczynski, 2012; Puissant, et al., 2013). Spanoudakis & Zisman (2001) define consistency as

> *"a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are jointly satisfiable"*

UML is the de-facto standard for modelling OO software systems (Huzar, Kuzniarz, Reggio, & Sourrouille, 2005; Puczynski, 2012). UML 2.3 defines 13 different diagrams. Relations between these diagrams are complex, and may lead to inconsistent UML diagrams (Liu, 2013; Torre, Labiche, & Genero, 2014). Coevolution among different perspectives or views of UML diagrams means that the modification in one diagram should be reflected to other related diagrams to ensure the consistency of all diagrams.

According to Lucas et al. (2009), the consistency problem in UML diagrams is linked to the multiple views of UML diagrams and the inconsistencies among these views or perspectives could be a source of numerous errors in the software developed which complicate diagrams management. If the effect of changes in UML diagrams is not addressed adequately among diagrams, it will result in further defects, decreased maintainability, and increased gaps between high-level design and implementation (N. Ibrahim, Ibrahim, Saringat, Mansor, & Herawan, 2013; Lehnert, 2011; Lehnert & Riebisch, 2013; Puczynski, 2012). Inconsistency problems could make the use of models as a source of automatic code generation impossible, such that the accuracy of generated code depends on UML models consistency (Simmonds & Bastarrica, 2005;

Usman, Nadeem, Kim, & Cho, 2008). As a summary of the main problems discussed in this section:

- Software models are highly dynamic and evolve from requirements through implementation. In order to respond quickly to varying requirements, it is extremely important to provide a change management technique to keep track of changes and to realise flexible and consistent software models.
- An OO modelling language describes a system by modelling different perspectives using its structural, behavioural, and interaction diagrams. The coevolution among these diagrams is high; therefore it is crucial to check the coevolution between the perspectives in these diagrams in order to control the change and keep these different views or perspectives consistent.
- UML as a standard language for modelling OO software systems is a semi-formal language and does not automatically support validation and verification of the coevolution between software models.

Hence, it is our concern to address the coevolution and inconsistency problems discussed in this section. Therefore, it is the aim of this research to propose an efficient coevolution framework for supporting coevolution between UML diagrams. The proposed framework aims to keep track of changes in UML diagrams. This includes ensuring the consistency between UML diagrams, tracing the diagrams' dependency, and determining the effect of the change in these diagrams after each change operation.

1.2 Research Motivation

Coping with software changes is one of the major issues in software analysis and design. Providing a change management technique to manage the coevolution among software models is one of the popular research areas in software analysis and design due to their numerous applications, and to ensure the models correctness in response to changes on them (Williams & Carver, 2010). Solving the coevolution and inconsistency problems in software models especially UML diagrams is a highly active research in which a considerable research work has been done (Dubauskaite & Vasilecas, 2013; Puczynski, 2012; Puissant, et al., 2013). However, there are important gaps and limitations still open for research.

Although the previous approaches in the state-of-the-art research provide solutions to handle software changes in UML diagrams, these approaches are concerned with some of the UML diagrams (i.e. the class, sequence, and statechart diagrams) and concentrate on checking the consistency by comparing two different versions from the same model. Additionally, there are limitations in managing the coevolution after adding, modifying, or deleting new models or diagrams or diagram elements. There is a need to handle the coevolution between UML diagrams perspectives and ensuring the consistency of all diagrams comprehensively using all UML structural, behavioural, and interaction diagrams including the diagrams relations.

Therefore, this research proposes a coevolution framework to cover the limitations discussed about coevolution and consistency of UML diagrams. A formal modelling language based on CPNs is used to model and simulate the proposed framework. The rational of using CPN stems from the fact that it provides automatic validation and verification. Formal methods improve software development specification, verification and validation, and this is very important for UML diagrams consistency analysis. According to Wordsworth (1999),

> *“a formal method of software development is a process for developing software that exploits the power of mathematical notation and mathematical proofs”*.

1.3 Research Objectives

The primary goal of this research is to enhance the representation capabilities of OO and CPNs modelling languages to support model changes in a rapidly changing environment. More specifically, this research aims to propose an efficient coevolution framework to trace dependency and to manage the coevolution between UML diagrams after each update operation, where UML diagrams are modelled from different perspectives using UML structural, behavioural, and interaction diagrams. In order to accomplish this primary goal, the following Research Objectives (RO) are outlined:

RO1: To propose a new structure for the integration of UML and CPNs (Object Oriented Coloured Petri Nets (OOCPNs) including the transformation rules applied between UML diagrams' elements and OOCPNs.

RO2: To propose a set of change impact and traceability analysis templates for the types of change in UML 2.3 diagrams, including rules to maintain consistency and integrity.

RO3: To propose a set of coevolution patterns to model and simulate the proposed diagrams changes. This includes the change impact and traceability analysis templates for updating UML diagrams.

RO4: To propose a coevolution framework based on the proposed structure, templates, and patterns.

RO5: To validate and verify the proposed framework, checking the correctness and performance analysis of the proposed coevolution framework.

1.4 Research Scope

This research focuses on proposing a new coevolution framework to manage the coevolution between software artefacts especially UML diagrams. This research proposes, develops, and implements a coevolution framework for UML diagram changes. In this capacity, the research covers issues related to changes to the elements of the diagrams in general, and includes a set of coevolution patterns, change impact and traceability analysis templates, and UML to OOCPNs transformation rules.

The idea of proposing a new structure for the integration between UML and OOCPNs is to integrate the proposed change impact, traceability analysis templates, and UML diagrams consistency rules into the transformation rules.

The proposed set of templates and the transformation rules into OOCPNs are defined for UML diagrams supported in UML 2.3. The proposed OOCPNs structure and the proposed templates cover all the UML diagrams provided in UML 2.3.

The proposed coevolution patterns are applied into the following UML diagrams (class, object, activity, statechart, and sequence diagrams). These diagrams cover the three perspectives of UML diagrams (structural, behavioural, and interaction). Several studies such as (Langer, Mayerhofer, Wimmer, & Kappel, 2014; Reggio, Leotta, Ricca, & Clerissi, 2013) mentioned that the class, activity, statechart, and sequence diagrams are the mostly used diagrams in UML analysis and design. Additional patterns for change control and management are also provided in the proposed framework. The relations between these patterns are identified and stated clearly.

1.5 Research Questions

In order to achieve the research objectives, the following Research Questions (RQ) are formulated to guide the research.

RQ1: How to integrate between UML and CPNs in order to perform diagrams coevolution?

RQ2: How to formulate the diagram changes in a patterns and templates design?

RQ3: How to provide an efficient coevolution framework for the coevolution between UML models in order to improve their flexibility to dynamic changes in a rapidly changing environment?

RQ4: How can the performance of the proposed coevolution framework are quantified?

1.6 Organisation of the Thesis

This chapter provides the context of the thesis along with the research motivation. In addition, research problem statement, research motivation, research objectives and questions, and research scope are identified and stated. The rest of the thesis is organized as follows:

The second chapter presents a literature review for this research. This chapter is on the theory building part of the research. A literature review on various concepts about software modelling and change management concepts is presented. Then, the findings from the literature review are summarized and the research direction is presented. This chapter surveys previous literature studies relevant to the field of study.

Chapter three is concerned with the proposed research methodology. The process of selecting the research idea, determining the research problem and objectives, formulating the research design, collecting and analysing the research data are discussed.

Chapter Four is concerned with the proposed coevolution framework to support detecting and resolving the coevolution and inconsistencies among OO diagrams. The proposed framework components and features are presented including a discussion about the research design and research procedures adopted.

Chapter Five is dedicated to provide the proposed structure for the transformation of UML diagrams into OOCPNs. This chapter discusses the proposed transformation rules of UML diagrams’ elements into OOCPNs. Additionally, the integration of these rules with the proposed change impact and traceability analysis templates is identified.

Chapter Six presents the proposed coevolution patterns to be applied to trace the dependency and to determine the effect of change between UML diagrams’ elements. In addition, patterns foundation, relations, and analysis are also identified. Additionally, the simulation methodology, scenarios, and results are discussed.

Chapter Seven is dedicated to the framework analysis, discussion of results, and performance analysis. The proposed framework is evaluated and compared to other approaches considering a wide range of performance parameters and metrics. The purpose of this chapter is to discuss the research findings.

Chapter Eight summarizes the thesis findings and highlights main contributions of this research. Finally, conclusions are drawn and suggested recommendations for some potential future research areas are highlighted.

CHAPTER 2: LITERATURE REVIEW

In this chapter, the results of a review of the literature on various topics related to the proposed framework are provided. Approaches and studies related to software change management, coevolution, and software modelling languages especially UML and CPNs are discussed. A summary of the main findings is also provided.

2.1 Software Change

Software change is a strategy-driven organizational initiative to improve and redesign processes to achieve competitive advantage in performance (Stemberger, Kovacic, & Jaklic, 2007). There are many reasons for changes in software models, for example, change of enterprise goals, change of client needs, and technological innovations (Tripathi, Hinkelmann, & Feldkamp, 2008). Change, according to Koomsub (1999), includes effects associated with strategy, structure, system, style, staff, shared values (or subordinate goals), and skill. Change also occurs frequently when the specification at design time is incomplete or when exceptional situations occur during execution (Capra & Cazzola, 2007). The nature of the change could be corrective, evolutionary, or ad hoc (Nurcan, 2008; W. Van Der Aalst, 1999). Corrective changes are implemented to correct a design error or to react to an exception that happens during execution. Evolutionary changes are required due to the redesign or reconfiguration of models. Ad hoc changes are related to non predefined actions.

Software change management is essential in Information Technology (IT) organizations and enterprises. Some approaches for managing the software change life cycle are provided in (Ghosh, Sharma, & Mohabay, 2011a, 2011b, 2011c) and Bhat and Deshmukh (2005). As a summary of these approaches, the main stages in software change management are: understanding the changed elements that are impacted, redesigning, and implementation.

An efficient mechanism for controlling and managing versions is required in a change management process. A version is *“a changed state of a specific target or concept from an existing state or condition”* (Kim, Kim, & Kim, 2007, p. 5). There are two types of versions: revision and variant (Kradolfer, 2000; X. Zhao & Liu, 2007). A ***revision*** is a version that is newly created by amending an existing version. A ***variant** is* a version that is created when two or more versions are derived from an existing version.

2.2 Software Coevolution

Understanding coevolution, which represents the dependency between artifacts that are frequently changed together, is important from the points of view of both practitioners and researchers (Jaafar, 2012). Coevolution is also considered a change propagation between diagrams at the same level of abstraction (Amar, Leblanc, Coulette, & Dhaussy, 2013). Maintaining coevolution and consistency between OO design elements could help practitioners to successfully perform their maintenance tasks (Hammad, Collard, & Maletic, 2010). Software changes are one of the main reasons for inconsistency problems in UML diagrams, where the change in one diagram element should be reflected in other diagrams. Spanoudakis & Zisman (2001) define consistency as:

> *“a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are jointly satisfiable”*(p.3).

Consistency is usually linked to the existence of multiple models or views that are involved in the development process (Engels, Küster, Heckel, & Groenewegen, 2001; Lucas, et al., 2009), and the set of activities for detecting and handling consistency problems is called inconsistency management (Lucas, et al., 2009).

Change impact analysis and traceability analysis are very important in solving the coevolution and inconsistency problems between UML diagrams. Software change impact is defined in (Bohner, 1996, 2002) as:

> *"The determination of potential effects to a subject system resulting from a proposed software change"* (p.265).

Change impact analysis identifies the scope of modifications that need to be implemented in response to a change (Jönsson, 2005). Traceability analysis is performed to analyse the dependencies between and across software artefacts at all levels of the software process (H. O. Ali, Rozan, & Sharif, 2012; S. Ibrahim, Idris, Munro, & Deraman, 2005). Dependency analysis and traceability analysis are the two primary methodologies for performing impact analysis (Kagdi, Gethers, & Poshyvanyk, 2012). The main difference between them is the level of abstraction. Dependency analysis analyses software artefacts at the same level of abstraction (e.g., source code to source code or design to design) and traceability analysis analyses software artefacts across different levels of abstraction (e.g., source code to UML) (Lam, Shankararaman, Jones, Hewitt, & Britton, 1998; Mohan, et al., 2008).

Traceability and consistency types are discussed in De Lucia, Fasano, and Oliveto (2008), Mens, Van Der Straeten, and Simmonds (2005a), and Usman, Nadeem, Kim, and Cho (2008). In summary, vertical traceability refers to the ability to trace dependent artefacts within a model, while horizontal traceability refers to the ability to trace artefacts between different models within the same version. Evolutionary traceability indicates the consistency between different versions of the same model. Meanwhile, semantic and syntactic consistency is based on the semantic meanings and specifications defined by the UML metamodel.

Change impact and traceability analysis approaches can be code-based or model-based (C.-Y. Chen & Chen, 2009; C.-Y. Chen, et al., 2007; Mahmood & Mahmood, 2015). Code-based impact analysis techniques require the implementation details of a change request or a precise change implementation plan prior to determining change impacts (C.-Y. Chen, et al., 2007). The approaches in Kung et al. (1994) and Weiser (1984) are code-based impact analysis techniques. Model-based impact analysis techniques identify and determine change impacts without using program code, and make proper decisions before considering any change implementation details (C.-Y. Chen, et al., 2007; Mohan, et al., 2008; Podgurski & Clarke, 1990). Model-based techniques identify change impacts by tracking the dependencies of software objects and classes within abstract models of the software design (C.-Y. Chen, et al., 2007). Control and data flow dependencies are the basic types of program dependencies (Podgurski & Clarke, 1990).

According to Lehnert (2011), assessment of model changes on a more abstract level than source code can enable impact analysis in earlier stages of development, which has become more important in recent years. Some approaches combine model-based and code-based change impact analyses. Examples of these approaches are presented in Murphy, Notkin, and Sullivan (1995) and Murphy, Notkin, and Sullivan (2001). Some studies on consistency management of UML diagrams and change impact analysis techniques are provided in Amar, Leblanc, Coulette, and Dhaussy (2013), Egyed (2006, 2011), Khalil and Dingel (2013), Li et al. (2012) , Lucas, Molina, and Toval (2009), and Stephan and Cordy (2013). Some approaches for the code-based and model-based change impact and traceability analyses are summarized in Table 2.1 and Table 2.2 respectively.

Table 2.1: Summary of Model-based Impact Analysis Techniques

Approach	Approach Description
C.-Y. Chen et al. (2007)	An approach for performing change impact analysis is presented to describe changeable items (objects, attributes, and linkages) and their relations and for tracking the dependencies of software objects and classes within abstract models of the software design.
Mohan et al. (2008) Park et al.(2009)	A process slicing approach to find change impacts in processes and activities is discussed. The process slicing approach is designed to formally operate on the software process by considering multiple perspectives such as behavioural, informational, and organizational perspectives. The traceability check is based on software artefact relationships.
De Lucia et al. (2008)	The work analyses the role of traceability relations in impact analysis. Additionally, it analyses the impact based on the relations between different artefacts.
(Ekanayake & Kodituwakku, 2015)	UML class and sequence diagrams are translated into XML Metadata Interchange format and then an algorithm is applied to check the consistency among these two diagrams.
Reder and Egyed (2012).	The purpose of this research is to improve the performance of the incremental consistency check. It focuses on the parts that are affected by model change, not on how to validate design rules.
Ali et al. (2006)	This approach ensures the validity of the conceptual model (class diagram) at the design stage by using Object Constraints Language (OCL).
Ibrahim et al. (2013)	This work uses use-case-driven-based rules to ensure consistency of UML model using a logical approach.
Egyed (2006, 2011), Elaasar and Briand (2004), and Millan, Sabatier, Le Thi, Bazex, Reder and Egyed (2013), Percebois (2009)	These works aim to ensure consistency between UML diagrams by using OCL.
Shinkawa (2006)	This approach involves a consistency check between use case, activity, sequence and statechart diagram using CPNs.
Gongzheng and Guangquan (2010)	This approach checks the consistency between state chart and sequence diagrams in UML 2.0. XYZ/E formal language (Tang, 2002), which is based on temporal logic (Pnueli, 1977), is used in the consistency check.
Isaac and Navon (2013)	Graph-based algorithms are used to identify which elements are affected by a change.

Approach	Approach Description
Puissant et al. (2013)	An artificial intelligence technique using both generated models and reverse-engineered models of varying sizes is employed to resolve the inconsistencies in UML models.

Table 2.2: Summary of Some Code-based Change Impact Analysis Techniques

Approach	Approach Description
Weiser (1984)	A process slicing approach is used to find the change impact in processes and activities.
J. Zhao (2002) (Bishop, 2004)	A program slicing technique is used to determine the change impact.
Xing and Stroulia (2005)	This work analyses the design evolution of OO software from the logical view using Java programming. This research focuses on detecting evolutionary phases of classes.
Gethers et al. (2012)	This work performs impact analysis from a given change request to source code.
Costanza (2001), Malabarba et al.(2000), Vandewoude and Berbers (2002)	Theses approaches uses runtime updates based on Java programming.
Huang and Song (2007)	Java programming is used to perform dependency analysis between OO entities.
Kagdi, Gethers, and Poshyvanyk (2012)	Both conceptual and evolutionary techniques are used to support change impact analysis in source code.
X. Sun, Li, Tao, Wen, and Zhang (2010)	This work analyses the impact mechanisms of different change types in Java programming.
Torchiano and Ricca (2010)	Source code comments and change logs in the software repository are used to analyse change impacts.
Kung et al. (1994) Zalewski and Schupp (2006)	This work concerned with code changes in OO libraries

In the following subsections (sections 2.2.1 to 2.2.4), some approaches from the literature on UML diagram coevolution, the inconsistency problem, and change impact and traceability analysis are reviewed and discussed. These approaches are classified into direct, transformational, formal semantics, or knowledge representation approaches. Hybrid approaches combine two or more of the above approaches (Khalil & Dingel, 2013). In Section 2.2.5, the diagramming tools that support coevolution and consistency management are reviewed. Some comments and discussions about these approaches are provided in each section.

2.2.1 Direct Approaches

Object Constraints Language (OCL) is used in many approaches to ensure consistency between UML diagrams as shown in Table 2.1 and Table 2.2. OCL is considered to be a direct approach for checking consistency such as it is integrated in some modelling tools (D. Chiorean, Paşca, Cârcu, Botiza, & Moldovan, 2004; D. I. Chiorean, Petrascu, & Petrascu, 2008; Sapna & Mohanty, 2007). Some approaches that use OCL to ensure consistency between UML diagrams are proposed in Egyed (2006, 2011), Ali et al. (2006), Elaasar and Briand (2004), Vasilecas, Dubauskaitė, and Rupnik (2011), and Millan, Sabatier, Le Thi, Bazex, and Percebois (2009). However,

Standard OCL does not allow making changes to the model elements to resolve them (Khalil & Dingel, 2013). Furthermore, CPNs can be used for checking and verifying the UML model associated with OCL to check whether it meets the user requirement or not (Sharaff, 2013).

In Briand et al. (2003) and Briand et al. (2009), an automatic change impact analysis technique is developed to detect the changes between two different versions of a UML model automatically. The UML model is composed of class, sequence, and statechart diagrams. In addition, consistency rules, which are formalized using OCL, are proposed. Horizontal and vertical traceability analyses are supported in this approach. *This approach is concerned with keeping the software models in a consistent state and synchronized with the underlying source code (Lehnert, 2011).* The following approaches summarized the set of consistency rules between UML diagrams from the literature (Briand, et al., 2003; Briand, et al., 2009; DAMIANO, LABICHE, & GENERO, 2015; Torre, 2015) .

In Egyed (2006, 2007a, 2007b, 2011), the change impact scope is determined based on a set of proposed consistency rules for UML class, sequence, and state diagrams. UML/Analyser and Model/Analyser tools (Egyed, 2007b) are developed to automate and evaluate the approach. A novel approach for improving the performance of incremental consistency checking was proposed in Reder and Egyed (2012). *The basic idea of this approach is to focus on the parts that are affected by model changes and not to validate design rules in their entirety.*

The purpose of the research in Ali, Boufares, and Abdellatif (2006) is to emphasize the importance of the global coherence of constraints in order to ensure the validity of the conceptual model (class diagram) at the design stage. The authors classify the integrity constraints that can be held in UML class diagrams from the conceptual perspective. Some of these constraints are OCL constraints, intra-association constraints, and interclass constraints (generalization, composition, and functional dependency constraints).

2.2.2 Transformational Approaches

The coevolution of OO software design and implementation approach is proposed in D'Hondt, De Volder, Mens, and Wuyts (2002) and Wuyts (2001). Logic metaprogramming is proposed as a way to affect a bidirectional link between software design and implementation. The automated coevolution of models using traceability analysis based on model transformation to code is proposed in Amar et al. (2013).

A coevolution approach between a component-based architecture model and OO source code is proposed in Langhammer (2013). *The coevolution in this approach is based on bidirectional mapping rules between architecture model and source code.*

García, Diaz, and Azanza (2013), Cicchetti, Di Ruscio, Eramo, and Pierantoni (2008), Wachsmuth (2007), and Hößler, Soden, and Eichler (2005) discuss the coevolution between metamodels and models based on model transformation to metamodels. *In these approaches, new updates are stored in a new version from the metamodel.* According to Protic (2011), model coevolution describes the problem of adapting models when their metamodels evolve.

Tracing model changes through a model synchronization approach is proposed in Ivkovic et al. (2004) to achieve traceability consistency. In this approach, models are transformed to use a graph metamodel. The transformed metamodel is then used to code model dependencies while equivalence relations are used to evaluate model synchronization. A change in a model is viewed as a combination of graph changes. A graph transformation approach is defined in Fryz and Kotulski (2007) to check the consistency between use case and class diagrams.

In Mens et al. (2005a), horizontal and evolutionary consistency rules between the UML class, sequence, and statechart diagrams are classified. In addition, the authors describe an extension to the UML metamodel to support the UML diagrams' versions. The authors discuss the importance of traceability analysis and change propagation in UML diagrams but they provide no support for this (Herzig, Qamar, Reichwein, & Paredis, 2011; Mäder, Gotel, & Philippow, 2009).

A tool for synchronous refactoring of UML activity diagrams using model-to-model transformations is presented in (Einarsson & Neukirchen, 2012). Refactoring is applied to improve the internal structure of source code (Fowler, 1999).

An evolution process at the requirement level based on the concept of gap analysis is proposed in Etien, Rolland, and Salinesi (2004) and Salinesi, Etien, and Wäyrynen (2004). *The proposed evolution process is applied in the context of organizational change*. A metamodel and a generic typology of operators are used to express different kinds of evolution.

The approaches in Costanza (2001) and Malabarba, Pandey, Gragg, Barr, and Barnes (2000) are examples of runtime updates based on Java programming, where (Malabarba, et al., 2000) focus specifically on dynamic Java. The authors extend the default Java class loader in such a way that class definitions can be replaced and objects or dependent classes can be updated. The replacement is initiated by the user through explicit calls to the class loader in the application program. In Costanza (2001), interface changes are allowed and do not require the application to be developed with evolution in mind based on dynamic delegation with Lava (a variation of Java). However, according to Spanoudakis and Zisman (2001), one drawback of this approach is the state space explosion problem.

Also according to (Puissant, 2012), the graph transformation technique is limited to check the structural inconsistencies only because it detects and resolves the inconsistencies which can be expressed as a graph structure only. Other approaches in consistency and coevolution based on transformational models are presented in other studies (Dang & Gogolla, 2016; Demuth, Riedl-Ehrenleitner, Lopez-Herrejon, & Egyed, 2016; Khan & Porres, 2015; Kusel et al., 2015).

2.2.3 Formal Semantics Approaches

In this subsection, some approaches that develop formal semantics in order to ensure the consistency and correctness of UML diagrams are discussed. Additionally, this research contains a complete study on formal approaches that use CPNs to check the consistency and correctness of UML diagrams which is discussed in Section 2.4.

A comprehensive survey of UML diagrams' change impact analysis techniques is discussed in Lucas, Molina, and Toval (2009). One of the findings of their survey is that formal languages are highly used to support detecting and determining the consistency and change impact between software models.

A CPNs approach to check the consistency of sequence diagrams with the system requirements is presented in (Ouardani, Esteban, Paludetto, & Pascal, 2006). In this approach, a technique for sequence diagram to Petri Nets (PNs) transformation is presented for the purpose of requirements validation and verification.

Shinkawa's (2006) approach requires a transformation from UML diagrams to other notations (CPNs) before checking the consistency. A framework for the verification of UML behavioural diagrams using PNs is proposed in Guerra and de Lara (2003) in which UML statecharts, activity, and collaboration diagrams are transformed to PNs for verification.

In Gongzheng and Guangquan (2010), XYZ/E formal language (Tang, 2002) which is based on temporal logic (Pnueli, 1977), is used to check the consistency between statechart and sequence diagrams in UML 2.0. A formal approach using graph grammars to check the consistency of UML class and sequence diagrams is proposed in (Tsiolakis & Ehrig, 2000).

According to N.C. Russell (2007), although more widely used as a systems modelling technique, UML is also suitable for business process modelling where it can capture the dynamic aspects of process modelling such as use case, activity, sequence, and statechart diagrams. However, UML has no formal basis to describe how these models can be integrated in order to provide a comprehensive view of a business process.

Formal languages such as Object-Z and CSP (Rasch & Wehrheim, 2003), have been used to check the consistency between UML class and statechart diagrams. B formal method is focused on refinement to code in checking the consistency between UML diagrams Osami, et al. (2005), where refinement means "*describing the new definitions of some parts of the specification's elements according to the required changes*" (Ossami, et al., 2005).

Formal approaches are widely used in describing the behaviour of UML diagrams using the executional model's capability provided in CPNs.

2.2.4 Knowledge Representation Approaches.

Knowledge-based approaches for OO diagram consistency checking are discussed in Calì, Calvanese, De Giacomo, and Lenzerini (2002), Baader (2003), and Bolloju et al. (2012). A knowledge-based system methodology to verify the consistency of a given object model against a set of use cases (defined as a natural language narrative) is proposed in (Bolloju, et al., 2012). In this methodology, missing and invalid diagram elements are identified to help the analyst create object models that are consistent with the requirements identified in the use case narratives. The use of use-case-driven-based rules for ensuring consistency in the UML model approach is proposed in Ibrahim et al. (2013). In Van Der Straeten, Mens, Simmonds, and Jonckers (2003), the UML metamodel and user-defined models are transformed into descriptive logic to check for consistency.

2.2.5 UML Diagramming Tools Support

A case study was undertaken by Amba (2009) to evaluate four management tools (IBM Rational RequisitePro, Borland CaliberRM, TopTeam Analyst, and Telelogic DOORS) in supporting change impact and traceability analysis. *This study indicates all* *these tools have poor impact analysis features. This shows that impact analysis in these management tools is very limited and thus more effective methods are needed.*

Some UML diagramming tools, such as the Visual Paradigm tool, detect the impact analysis based on the physical connection between the elements of UML diagrams. *The Visual Paradigm tool analyses the connection between the diagrams' elements based on the user selection for the dependency between the diagrams.* The ArgoUML tool detects incremental consistency checks in UML diagrams, but it requires annotated consistency rules (Egyed, 2006). According to Tam et al. (2000), the Rational Rose diagramming tool provides change management by transforming a diagram into a hierarchical text description and highlighting the changed items within the transformed text.

A set of rules to check consistency between UML diagrams is identified in (Liu, 2013). These rules are helpful for developers who need to check the consistency between class, activity, statechart, sequence, and communication diagrams. The author discusses methods of applying these consistency rules. These methods are: manual, compulsory restriction, automatic maintenance, and dynamic check.

As a summary for the coevolution appraches discussed in this section, the direct approaches use the constructs of OO and Object Constraints Language OCL, transformational approaches derive a common notation by transforming one model to another. Formal approaches develop formal semantics for the OO diagrams, while knowledge representation approaches use description logics as a representation language. A hybrid approach is a combination between two or more different type of these approaches (Khalil & Dingel, 2013).

2.3 Patterns

A pattern *"describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* Alexander et al. (1977, p. 256). According to NA Mulyar (2009, p. 1), Nataliya Mulyar and van der Aalst (2005), and Weber, Rinderle, and Reichert (2007),

> *"Pattern languages are based on experience; they express sound solutions for problems frequently recurring in a certain domain in a pattern format".*

A pattern language helps developers to build efficient models by avoiding the reinvention of already existing solutions to problems. Software models and patterns can be integrated together in software development because patterns can be used as templates for software development models (Côté & Heisel, 2009). Additionally, patterns enhance the software structure by decoupling different components and this makes the evolution tasks easier. In OO, design patterns make it easier to reuse successful designs and architectures (Gamma , Helm, Johnson, and Vlissides (1995), Gamma, Helm, Johnson, and Vlissides (2001), and Meijers (1996)). The following is a definition of a pattern proposed by Alexander (1979):

Pattern name: *The identifier of a pattern which captures the main idea of what the pattern does;*

Also known as: *An alternative name for the pattern name;*

Intent: *describes in several sentences the main goal of a pattern, i.e. the problem for which it offers a solution;*

Motivation: *Describes the actual context of the problem addressed and why the underlined problem needs to be solved.*

Problem description: *Presents the problem addressed by the pattern;*

Solution: *Describes possible solutions to the problem;*

Implementation of solution: *Illustrates how to implement the described solution;*

Applicability: *The typical situations in which the pattern can be applied;*

Consequences: *Outlines the possible advantages/disadvantages of using the pattern; in cases where the pattern supplies several solutions, this section elaborates on the differences between them;*

Examples: *Lists several examples demonstrating the use of the pattern in practice;*

Related patterns: *Specifies relations between the pattern and other patterns.*

Gamma, et al (Gamma , et al., 1995) and Gamma, et al (Gamma, et al., 2001) modify the pattern definition proposed by (Alexander, 1997) for use in OO software design. The modified pattern is as follows:

Intent: *What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?*

Motivation: *A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.*

Applicability: *What are the situations in which the pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?*

Participants: *The classes and/or objects participating in the design and their responsibilities.*

Collaborations: *How the participants collaborate to carry out their responsibilities.*

Diagram: *A graphical representation of the pattern using a notation based on the object modelling techniques.*

Consequences: *How does the pattern support its objective? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?*

Implementation: *What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are these language-specific issues?*

Example: *Examples of the pattern found in real systems.*

See Also: *What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?*

Patterns are used in many workflow software systems to manage and execute operational processes involving people, applications, and/or information sources on the basis of process models. These activities-based patterns are divided into general patterns, workflow control flow patterns, service interaction patterns, and process flexibility patterns (NA Mulyar (2009), Nataliya Mulyar and van der Aalst (2005), and (Weber, Sadiq, and Reichert (2009)). Some of these patterns are modelled and simulated using CPNs as in Nataliya Mulyar and van der Aalst (2005). Pattern language verification in the model-driven design approach is introduced in (Zamani & Butler, 2013). A pattern language for evolution reuse in component-based software architectures approach is proposed in (Abbasi, 2015).

As reviewed in this section, patterns are used in two main areas of software modelling: as design patterns and in the workflow software management system.

2.4 Integration of UML and CPNs

In this section, the benefits derived from the integration of UML and CPNs in supporting software model coevolution and consistency checks are discussed. In addition, the integration techniques and approaches are provided.

The use of UML diagrams as OO diagramming techniques has become extremely popular because it offers powerful structuring facilities that place an emphasis on encapsulation and promote software reuse; however, this approach remains semi-formal and still lacks tools for automatic validation (Bousse, 2012). CPNs modelling language is used for the formal specification. CPNs have a natural graphical representation, which aids in the understanding of formal specifications and a range of automated and semi-automated analysis techniques. However, the weakness of CPNs formalisms is their inadequate support for compositionality, which means there is a need to provide structuring facilities, encapsulation and inheritance (Charles Lakos, 2001).

The integration of OO and CPNs formalisms is crucial in enabling software engineers and organizations to reap the complementary benefits of these two paradigms. The main advantages that can be gained from the integration of OO and CPNs modelling languages are the effective combination of the best characteristics of CPNs and OO design methods and better representation of system complexity as well as ease in adapting, correcting, analysing, and reusing a model (Chukwuogo, 2007; Kurt Jensen & Kristensen, 2009; Kurt Jensen, et al., 2007; Lewis, 1996; Mikolajczak & Sefranek, 2003). According to Bastide (1995), the three directions for integrating the PNs and OO concepts are:

a. **Integration of OO concepts into PNs:** PNs control the overall dynamic behaviour of a system, while ‘tokens’ represent objects that model the system’s static properties, as shown in Figure 2.1. The LOOP (Charles Lakos & Keen, 1994; Charles Lakos, Keen, & Hobart, 1991), Macronet (Keller, Shen, & Bochmann, 1994), and SimCon (Verkoulen, 1994) are examples of the integration of OO concepts into PNs.

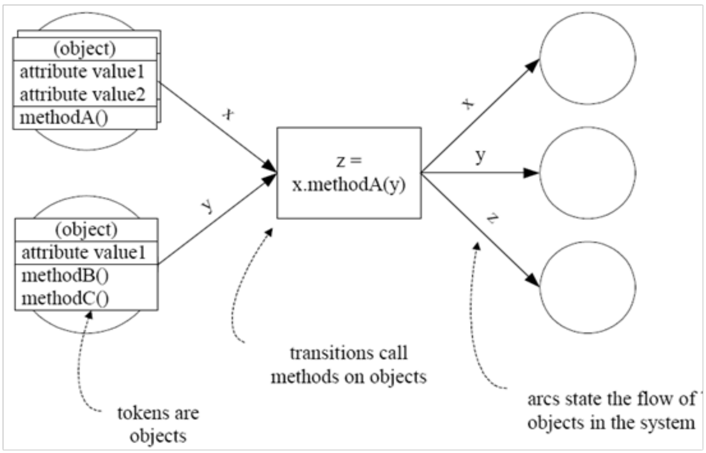

Figure 2.1: Integration of OO Concepts into PNs Bastide (1995, p. 1)

b. **Integration of PNs into OO techniques:** Here, a system is structured with OO techniques. First, the relevant objects of the discourse world and their mutual relationships are identified. Then, the description of the object behaviour and the communication between objects is specified with the help of PNs (Zapf & Heinzl, 1999), as shown in Figure 2.2. The OOBM (Hanish & Dillon, 1997) is an example of the integration of PNs into OO techniques.

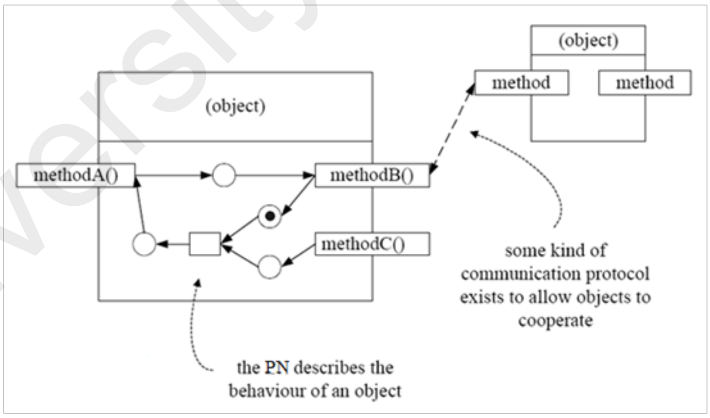

Figure 2.2: Integration of PNs into OO Techniques (Bastide (1995, p. 2))

c. **Mutual integration of OO techniques and PNs:** This approach is perceived as a further development in embedding PN models into objects. Here, objects are initially used to determine the structure of a system. Subsequently, the behaviour of the objects is modelled with the help of nets (Zapf & Heinzl, 1999), as shown in Figure 2.3. The OOPNL (Esser, 1997) and COOPN/2

(Biberstein, Buchs, & Guelfi, 1996) are examples of the mutual integration of OO techniques and PNs.

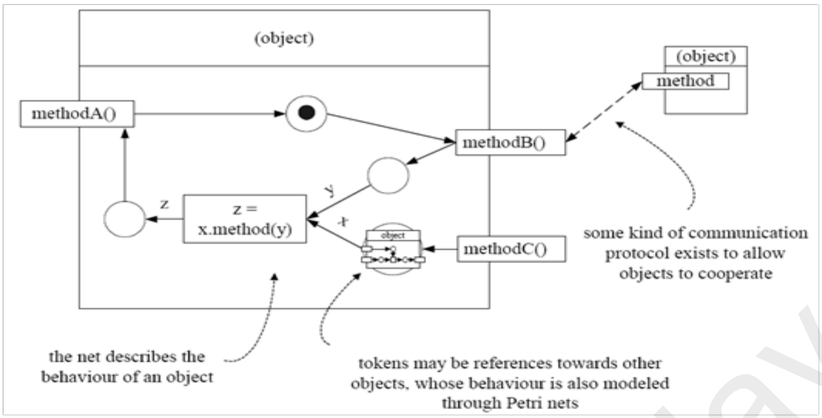

Figure 2.3: Mutual integration of OO Techniques and PNs (Zapf and Heinzl (1999, p. 10))

According to Tadj and Laroussi (2005), the representation of objects in CPNs is as follows: object classes and states classes are represented by places; object instances are represented by tokens; and the object value state is represented by function. Object Oriented Petri Net (OOPN) modelling is a collection of elements comprising constants, variables, net elements (places and transitions), class elements (object nets, method nets, synchronous ports, and message selectors), classes, object identifiers, and method net instance identifiers.

An OOPN has an initial class and initial object identifier as well. The so-called universe of an OOPN contains (nested) tuples of constants, classes, and object identifiers (Krena & Vojnar, 2001). An OOPN is applied in different domains (Zapf & Heinzl, 1999), for example, in technical computer science (in modelling and simulation of distributed and concurrent systems, modelling of network protocols, real-time and embedded systems), in software engineering (in modelling of graphical user interfaces, design of database applications, and prototyping of OO design models), and in information systems (in enterprise modelling, office information systems, workflow systems and automation techniques). A framework to transform UML statecharts and collaboration diagrams into CPNs is proposed in Hu and Shatz (2004) to provide a dynamic model analysis. In this approach, statechart diagrams are converted into CPNs, and collaboration diagrams are used to connect the statecharts into a single CPN model.

Object PN Models (OPMs) (Saldhana & Shatz, 2000) are used to generate a Petri Net (PN) model from a UML object diagram. In this approach, object classes and state classes are represented using CPN places, while object instances are represented by CPN tokens. The generation of object CPNs from UML statechart diagrams is proposed in Bokhari and Poehlman (2006). An abstract node approach is used to transform an OO model into a hierarchical CPN model (Bauskar & Mikolajczak, 2006). Using this approach, class and sequence diagrams can be transformed to CPNs.

The transformation of UML 2.0 sequence diagrams into CPNs is presented in Fernandes et al. (2007) and (Khadka, 2007). The aim of the approach presented by Shin et al (Shin, Levis, & Wagenhals, 2003; Shin, Levis, Wagenhals, & Kim, 2005) is to model the transformation of the UML use case, class, and collaboration diagrams to CPN models. The integration of OO design with CPNs is developed by Motameni et al. (2008) for analysis purposes. In their work, the CPN model is used to verify the UML diagrams before implementation. The metamodelling and formalism transformation framework proposed by (Guerra & de Lara, 2003) is a general framework for the analysis of software systems using model-checking. This framework transforms the UML model into PNs for further analysis. The UML model is composed of classes, statecharts, and sequence diagrams.

A hierarchical OOPN integrates hierarchical PN with OO concepts to support OO features including abstraction, encapsulation, modularization, message passing, inheritance, and polymorphism (Hong & Bae, 2001; Xiaoning, Zhuo, & Guisheng, 2008). A metalevel and highly automated technique based on a graph transformation approach is presented in Zhao et al. (2004). This approach formally transforms UML statecharts and behavioural diagrams into PNs for verification.

A methodology to derive CPNs from UML object, sequence, statechart, and collaboration diagrams is proposed in Bouabana-Tebibel and Belmesk (2004, 2005). Some of the PN modelling languages adapt the OO concepts in PN and are called OOPN, as in Niu, Zou, and Ren (2003). The main concepts upon which these approaches are based are as follows: the OOPN is a set of class nets; a class is specified by a set of object nets, method nets, synchronous ports, negative predicates, and message selectors; object nets and method nets can be inherited; and a token represents an object or instance of class. Synchronous ports are special transitions which cannot fire alone; they are only dynamically fused to some regular transitions.

The approach to integrate OO design with CPNs was developed by Bauskar and Mikolajczak (2006) and Motameni et al. (2008) to check the correctness of the designed system. The approach integrates OO techniques at the design level and uses CPNs at the verification and validation level. The approach includes a technique to transform an OO design into a hierarchical CPNs model by using the abstract node approach (Bauskar & Mikolajczak, 2006).

The Object Oriented Petri Nets with Modularity (OOMPNets) model (Wang & Wang, 2007) is an advanced CPNs model that introduces CPNs into OO techniques. In this approach, the analysis techniques based on CPNs can be applied to reduce the effects of specification errors. The OOMPNets model supports gradual progress in modelling software requirements with formal representation of the actor, data views, control flow, and data flow. The incomplete specifications are encapsulated in nodes with hierarchical presentation to support forward and backward traces. The flexibility to present incomplete specifications in a formal format can allow the analysis of these specifications by those techniques used in CPNs. More approaches for transforming

UML structural, behavioural, and interaction overview diagrams are provided in Campos and Merseguer (2006), Jørgensen (2003), Liles (2008), Merseguer and Campos (2003) , and Miller (2003).

A comparative study of software tools that support the transformation of UML static and dynamic diagrams to PNs/CPNs models is presented in Rajabi and Lee (2009b). Some of these tools are ArgoSPE (Gómez-Martínez & Merseguer, 2006) and WebSPN (WebSPN-Research-Group, 2009). A summary of this comparative study is provided in Table 2.3.

Table 2.3: Representation Capabilities of Some Related Works in Transforming UML Diagrams to PNs and CPNs

Diagrams supported **Approach**	**Structural Diagrams**						**Behavioural Diagrams**			**Interaction Diagrams**			
	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
ArgoSPE (Gómez-Martínez & Merseguer, 2006)	√					√		√	√				
Calderon Prototype (Calderon, 2005)	√						√				√		
Chen (2000)	√						√	√					
Baresi (2002)	√								√		√		
Hu and Shatz (2004)									√		√		
Barros and Gomes (2004) Wang (2007), Watanabe et al. (1998), Shengyuan and Yuan (2007), and He (2000)	√												
Bokhari and Poehlman (2006)									√				
van der Aalst (2002)								√	√	√			
Guerra and de Lara (2003) and (Yao & Shatz, 2006)	√								√	√			
Abstract Node (2006)	√									√			
Lassen (2007)										√			
Shin et al. (2003), Barros and Jorgensen (2005)	√						√				√		
Elkoutbi (2000)	√						√			√			

Diagram supported Approach	Structural Diagrams						Behavioural Diagrams			Interaction Diagrams			
	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
Maqbool (2005), Liles (2008), Tričković (2000), Bouabana-Tebibel (2007), Garrido and Gea (2002) and Staines (2008)								√					
AMABULO(Bruckmann & Gruhn, 2008a; Brückmann & Gruhn, 2008b)	√							√	√				
Emadi and Shams (2008, 2009)					√		√			√			
Object Dynamics and Behaviour (Bouabana-Tebibel & Belmesk, 2004)		√							√	√	√		
OPMs (Saldhana & Shatz, 2000)		√											
Wagenhals, Haider, & Levis (2002, 2003)	√							√		√	√		

Note: CD: Class Diagram, OD: Object Diagram, PD: Package Diagram, CoD: Component Diagram, DD: Deployment Diagram, CSD: Composite Structure Diagram, UCD: Use Case Diagram, AD: Activity Diagram, SCD: Statechart Diagram, SD: Sequence Diagram, CommD: Communication Diagram, TD: Timing Diagram, and IOD: Interaction Overview Diagram.

2.5 Background on Software Modelling Languages

In this section, a brief background on software modelling languages is provided. First, the graph-based and rule-based modelling languages are reviewed. Second, UML diagrams and PNs/CPNs are discussed in detail.

Software modelling is one of the most important activities in software analysis and design. It provides a high-level specification independent from the implementation of such a specification. Software models can be evolved into a new version, and can be used to generate executable code (Van Der Straeten, 2005).

Graph-based formalism and rule-based formalism are the two most predominant formalisms in the development of modelling languages. Graph-based formalism has its roots in graph theory or its variants, while rule-based formalism is based on formal logic (Lu & Sadiq, 2007). Graph-based languages have the visual appeal of being intuitive and explicit, even for those who have little or no technical background. However, rule-based modelling languages require a good understanding of propositional logic and the syntax of logical expressions, and thus, are less attractive from the usability point of view (Lu & Sadiq, 2007).

2.5.1 Graph-based Modelling Languages

In a graph-based modelling language, the process definition is specified in graphical process models, where activities are represented as nodes and control flow and data dependencies between activities are shown as arcs. The graphical process models provide an explicit specification for process requirements (Lu & Sadiq, 2007). Graph-based modelling addresses the need to present software models to various stakeholders in as straightforward a manner as possible (Kowalkiewicz, Lu, Bäuerle, Krümpelmann, & Lippe, 2008). The following are examples of graph-based modelling languages:

OO Methodology: An established technique for structured software design (Aguilar-Saven, 2004). It supports inheritance, polymorphism, and dynamic binding. It is useful for designing software that is comprehensible, maintainable, and flexible (Bauskar & Mikolajczak, 2006). One of the main advantages of the OO method is the effectiveness of the process in terms of identifying and refining objects (Aguilar-Saven, 2004). Techniques for OO analysis and design primarily support the representation and integration of static system properties from a function and data perspective. Dynamic properties are supported from a process perspective (Zapf & Heinzl, 1999). UML is used as a language for specifying, visualizing, constructing, and documenting the artifacts of OO software systems, as well as for business modelling (Bauskar & Mikolajczak, 2006; N. Russell, van der Aalst, Ter Hofstede, & Wohed, 2006).

UML Activity Diagram (OMG, 2004, 2010; N. Russell, et al., 2006): Designed for modelling business process and flows in software systems. It also provides a high-level means of modelling dynamic system behaviour (N. Russell, et al., 2006).

Business Process Definition Metamodel (BPDM) (OMG, 2004, 2010): The BPDM does not provide its own graphical notation, which is specified as a UML 2.0 profile. The BPDM is used to define a generic metamodel in order to support the mapping between different tools and languages. Business Process Modelling Notation (BPMN) (Owen & Raj, 2003) is designed for modelling business processes and transforming them into an execution language.

PN theory: Widely used in graph-based modelling languages (K Jensen, 1992; Kurt Jensen, 1994, 1998; TGIgroup, 2013). Places and transitions are the main components of a PN model, and arcs are used to connect them. The main characteristics of CPNs are data structures and hierarchical structures (K Jensen, 1992; Kurt Jensen, et al., 2007). These characteristics are used to represent the object dynamics and to check the model's correctness (Kurt Jensen & Kristensen, 2009; Kurt Jensen, et al., 2007; Michael

Westergaard & Verbeek, 2013). Object PNs extend the formalism of CPNs with OO features, including inheritance, polymorphism, and dynamic binding (Koci, Janousek, & Zboril, 2008; Liui, Yin, & Zhang, 2008; Miyamoto & Kumagai, 2005, 2007; Yu & Cai, 2006). Timed PNs, as the name implies, introduce time in PNs.

Flow charts, data flow diagrams, role activity diagrams, role interaction diagrams, and the integrated definition for function modelling are also approached from a graph-based perspective and are discussed in detail in Aguilar-Saven (2004).

2.5.2 Rule-based Modelling Languages

A rule-based language integrates complex process logic into a process model to support dynamic changes (Lu & Sadiq, 2007; zur Muehlen, Indulska, & Kamp, 2007; Zur Muehlen, Indulska, & Kittel, 2008). In a typical rule-based modelling language, process logic is abstracted into a set of rules, each of which is associated with one or more activities specifying the properties of the activity such as the pre and post conditions of execution (Lu & Sadiq, 2007).

There are several classification schemas for business rules. According to Halle and Ronald (2001), there are four kinds of business rules: constraint rules, action enabler rules, computation rules, and inference rules. Fuzzy business rules were added later, as described in Thomas, Dollmann, and Loos (2007). The following are examples of rule-based modelling languages:

- Event-driven Process Chain (EPC) (Knolmayer, Endl, & Pfahrer, 2000; Scheer, 1994, 2000): The basic elements of this modelling language are functions and events. Functions model the activities, while events are created by processing functions or by actors outside the model.
- Integrated Event-driven Process Chains (iEPCs): Basically, these extend EPCs by using formal concepts of object flow and a role perspective (Mendling, La

Rosa, & ter Hofstede, 2008). The main idea is to show how any of these formalizations can be enhanced with transition rules that consider object existence and role availability as part of the state concept.

- PLMflow (Zeng, Flaxer, Chang, & Jeng, 2002) and ADEPT system (Jennings et al., 2000): These both provide a set of business inference rules that is designed to dynamically generate and execute workflows.
- ConDec language (Pesic & van der Aalst, 2006): A declarative language to specify which tasks are possible. Users can execute such a model according to their own preferences; they can choose which tasks to execute and how many times, and in which order to execute them.

However, the rigidity of graph-based approaches leads to problems such as lack of flexibility when faced with dynamic changes and lack of adaptability, which compromise the ability of the graph-based processes to react to dynamic model changes and exceptional circumstances (Lu & Sadiq, 2007). On the other hand, the rule-based approach is intended to integrate complex process logic into a process model as rules in order to support dynamic changes. More approaches for graph-based and rule-based modelling languages are provided in Rajabi and Lee (2009a).

Workflow management tools enable the runtime system to assist users in coordinating and scheduling the tasks of a business process in workflow management systems (Hollingsworth & Hampshire, 1993) by adding, deleting, or changing the sequence of process executions during runtime. These approaches are based on activity-oriented approaches. OO approaches have comprehensive modelling constructs of object orientation to capture business processes (N. Russell, et al., 2006), where the processes are modularized along key business objects rather than activity decompositions (Redding, 2009). Some examples of these approaches are provided in (Weske (1998), (Dadam & Reichert, 2009; Manfred Reichert & Dadam, 1998, 2009; MU Reichert, Rinderle, Kreher, & Dadam, 2005), Sun and Jiang (2009), (Lu, 2008), Wörzberger et al. (2008), Milanovic et al. (2008), and Van Hee et al. (Grossmann, Mafazi, Mayer, Schrefl, & Stumptner, 2015; 2006)) .

2.5.3 UML Diagrams

UML diagrams are interrelated; some components for one diagram may be derived from other diagrams. UML 2.3, which is one of the most recent versions of UML (Barr & Pettis, 2007; Bennett, et al., 2010; OMG, 2010), supports a variety of diagrams to model software systems from different perspectives using UML structural, behavioural, and interaction diagrams (Fowler, 2004), as shown in Figure 2.4.

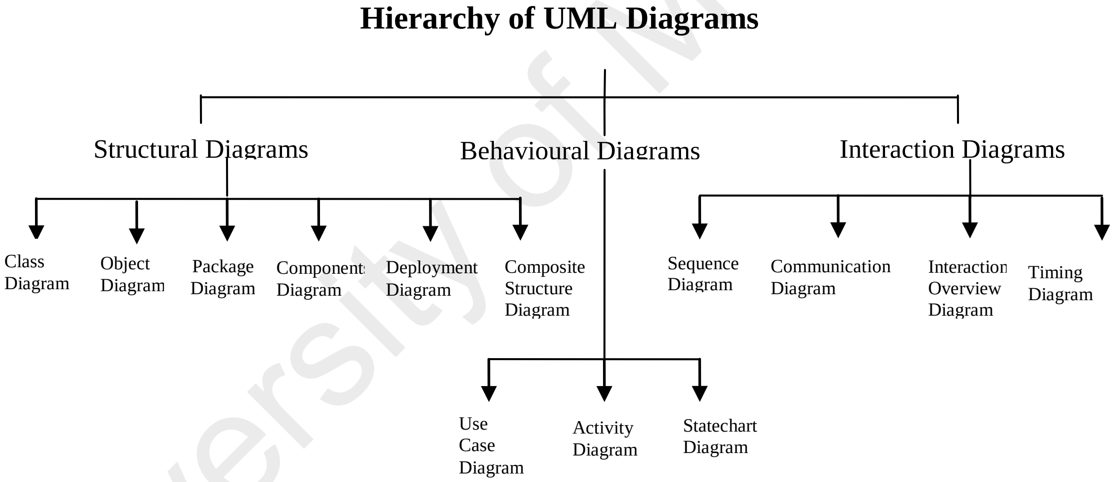

Figure 2.4: Hierarchy of UML Diagrams

The different perspectives of UML diagrams are discussed in the following subsections. Examples of UML diagram software tools are: Visual Paradigm (Curtis, Clarence, & Ying, 2005; VisualParadigmCompany, 2011), MagicDraw (MagicDraw, 2009), and IBM Rational Rose (IBMSoftware, 2011).

2.5.3.1 UML Structural Diagram Perspectives

Structural diagram perspectives are used to construct the information structure. These diagrams are briefly described below:

- A *class diagram* is useful to represent information about the actors, roles, organizational unit, and relevant data (Yang & Chen, 2003).
- Actors and data stores are objects in the *object diagram*.
- A *package diagram* organizes the diagram elements into related groups to minimize the dependencies between different diagrams' elements.
- A *composite structure diagram* can be used to show the internal structure and possible collaborations.
- A *component diagram* shows the dependencies among software components.
- A *deployment diagram* depicts a static view of the runtime configuration of the hardware nodes and the software components that run on those nodes (Miller, 2003).

2.5.3.2 UML Behavioural Diagram Perspectives

Behavioural diagram perspectives show how a system operates. These diagrams are briefly outlined below:

- The static interactions between diagrams and their external objects are expressed using a use case diagram (Yang & Chen, 2003). This type of diagram is used to express functionality, goals, and responsibility (C.-Y. Chen, et al., 2007).
- *An activity diagram* describes the dynamic behaviour of use cases. It is used to model the logical steps and the dynamic behaviour derived from the use cases (Chang, Chen, Chen, & Chen, 2000; Hongmei, Biqing, & Shouju, 2000). It concentrates on the dynamic relationships among business activities (Yang & Chen, 2003).
- A *statechart diagram* describes the process behaviour of states and events (Merseguer & Campos, 2003).

2.5.3.3 UML Interaction Diagram Perspectives

Interaction diagram perspectives can be considered a subset of behavioural diagrams. These diagrams are described in brief below:

- *Sequence diagrams* and *communication diagrams* are used to describe the interactions and flow of control among business objects based on messages. They represent the relationships between diagrams and actors. A sequence diagram focuses on the message times, while a communication diagram focuses on object roles. A communication diagram can be used to show the use case’s objects and the sequence of messages passed between them.
- An *interaction overview diagram* is a modification of the activity diagram that is used to compose interactions through sequence, iteration, concurrency, or choice concepts (Marzeta, 2007; Ribeiro & Fernandes, 2006).
- A *timing diagram* shows the behaviour of the processes in a given period of time; these diagrams could have a starting and finishing time to determine the sequence of activities or execution order.

2.5.3.4 Petri Nets and Coloured Petri Nets

Petri Nets are a powerful instrument for modelling, analysing, and simulating dynamic systems with concurrent and nondeterministic behaviour. They are useful for describing information systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic (Kurt Jensen & Kristensen, 2009). The graphical representation and executable nature of a PN model make the PN suitable for use in the simulation, rapid prototyping and verification of systems (Le Bail, Alla, & David, 1991). According to Aguilar-Saven (2004) and Murata

(1989), a PN is a directed graph that mainly consists of two different nodes: places and transitions, where places represent possible states of the system and transitions are events or actions that cause the change of state (Milanovic, et al., 2008; Scheer, 1994).

However, early attempts to use PNs in practice revealed two serious drawbacks (Aguilar-Saven, 2004). First, there were no data concepts and hence the models often became excessively large because all data manipulations have to be represented directly in the net structure. Second, there were no hierarchy concepts, and thus it was not possible to build a large model via a set of separate sub-models with well-defined interfaces. High-level PNs (HPNs) and Low-level PNs (LPNs) (Miyamoto & Kumagai, 2007; Wolf, 2009) are types of PNs. HPNs support abstract data types and state transitions with data processing, but LPNs do not have a data type and data processing mechanism. The choice of LPNs or HPNs depends on what kind of system is to be modelled. Generally, analysis of LPNs is comparatively easy, but a net of this type generally grows large. In contrast, HPNs can express a system in a compact net, but on the other hand, analysis of HPNs is difficult.

A CPN model (Aguilar-Saven, 2004; Kurt, 1997) incorporates both data structuring and hierarchical decomposition without compromising the qualities of the original PNs and thus removes these two serious problems that are inherent in PNs. Timed PNs (Holliday & Vernon, 1987) introduced time in PNs, while hybrid PNs (Le Bail, et al., 1991) can model a system where discrete state transitions and continuous state transitions coexist. CPN tools perform syntax and type checking as well as simulation code generation. More details about PNs theory, structure, and applications are provided in Kurt Jensen and Kristensen (2009) and Kordic (2008). A CPNs structure is defined formally as a set of **(∑, P, T, A, N, C, G, E, $\mathbf{M_0}$, I, O)** (Kordic, 2008; Kurt, 1997), where:

∑: A finite set of non-empty types, called a colour set
***P**: Finite set of places*

***T**: Finite set of transitions*

***A:** Represents a set of directed arcs, known as flow relationships. An arc exists between a place and a transition, or vice versa*

***N:** A node function*

***C:** A colour function*

***G:** A guard function defined from T into expressions*

***E:** An arc expression function defined from A into expressions*

M_0: *The initial (coloured) marking defined from P into closed expressions*

***I:** A function which determines the input multiplicity for each input arc*

***O:** A function which determines the output multiplicity for each output arc.*

2.6 Discussion and Summary

Making sure there is coevolution between the perspectives of UML diagrams and ensuring that there is consistency between all diagrams are important activities in software analysis and design. However, it is difficult to maintain coevolution and consistency between UML diagrams because these diagrams are continuously updated in order to reflect software changes. In this chapter, the researcher reviewed and discussed the approaches related to software change management, especially software models coevolution. The approaches that deal with solving the coevolution and inconsistency problems in UML diagrams and the approaches that address the integration between UML diagrams and CPNs were discussed in detail.

Detecting and resolving the coevolution between software artifacts can be achieved by using various techniques. Some of these techniques are: analysing release histories or versions, source code, and software architecture level analysis (Breivold, et al., 2012). These techniques can be classified into code-based and model-based approaches. Furthermore, assessments of model changes on a more abstract level than source code can enable impact analysis in earlier stages of development (Lehnert, 2011).

Decades of research efforts have produced a wide spectrum of approaches and techniques for checking the coevolution and inconsistency among OO diagrams. Some of these approaches can be classified into direct, transformational, or formal semantics approaches (Sapna & Mohanty, 2007). The main ideas and weaknesses of these approaches are: Standard OCL as a direct approach is concerned with keeping the software models in a consistent state and synchronized with the underlying source code and does not allow for making changes to the model elements to resolve them (Khalil & Dingel, 2013; Lehnert, 2011). CPNs can be used to check and verify the UML model associated with the OCL to ascertain whether or not it meets the user requirement (Sharaff, 2013). The coevolution in transformational approaches is based on bidirectional mapping rules between the architecture model and source code. The graph transformation technique is limited to checking the structural inconsistencies only because it can only detect and resolve the inconsistencies that can be expressed as a graph structure (Puissant, 2012). Formal approaches are widely used for describing the behaviour of UML diagrams using the executable model capability provided in CPNs.

As regards the usage of patterns in software modelling, researchers have concentrated on using patterns as design patterns and in the workflow software management system. Updating the pattern design to manipulate the software changes and change impact also could facilitate software change design. *Improving the effectiveness and the accuracy of state-of-the-art coevolution techniques in managing OO diagram changes is an important issue and much work is still needs to be done to fully provide flexibility, adaptability, and dynamic reaction to changes.*

Transforming UML diagrams into a formal modelling language such as CPN models is considered one of the most effective ways to solve software performance evaluation problems (Lian-Zhang & Fan-Sheng, 2012). The integration of UML and CPNs approaches is based on the combination of the best characteristics of the CPNs and UML design methods. While UML describes the static aspects of systems, the CPNs model system dynamics and behavioural aspects. The graphical representation and automated analysis techniques in CPN tools are used to aid the understanding of formal specifications (Barros & Gomes, 2004; Barros & Jorgensen, 2005; Niu, et al., 2003; Michael Westergaard & Verbeek, 2013). The transformation approaches discussed in this chapter have certain weaknesses, such that each transformation approach uses only a subset of UML diagrams, and most of these transformations are based on behavioural UML diagrams, as shown in Table 2.3. Additionally, these approaches focus only on a comparison between two versions from the same model to check if there are differences between them. *There is a need to support the change incrementally (i.e, during the design process and to also check the consistency between diagrams based on the diagram relations. Additionally, there is a need to support the changes by adding new diagrams).* The needs and details of the coevolution framework for this research are discussed in the following two chapters.

CHAPTER 3: RESEARCH METHODOLOGY

In this chapter, the general steps of the research methodology are outlined. This research methodology consists of several phases, as shown in Figure 3.1. These phases are:

- Research Idea Phase
- Literature Review Phase
- Research Design Phase
- Modelling and Development Phase
- Analysis and Evaluation Phase

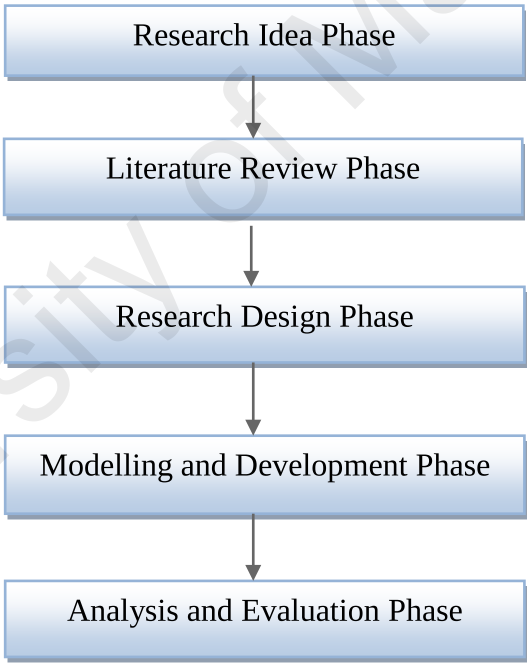

Figure 3.1: Phases of Research Methodology

3.1 Research Idea Phase

In this phase, the research idea is outlined. This includes the problem statement, research objectives, and research questions. The determination of the research problem, involves a few different stages, but mainly this research starts with the context of the research, which is the field of software change management, as shown in Figure 3.2.

This research focuses on studying the impact of software changes on modelling techniques and languages (basically on graph-based and model-based approaches) because it is one of the main issues in software design. OO software modelling is widely used in software modelling and design, and OO diagrams are divided into different perspectives for modelling a problem domain. This research focuses on determining the main issues that need to be addressed to preserve the coevolution among these diagrams so that they can be updated continuously to reflect software changes. In addition to these steps in determining the research problem, a clear statement of research objectives and research questions are defined.

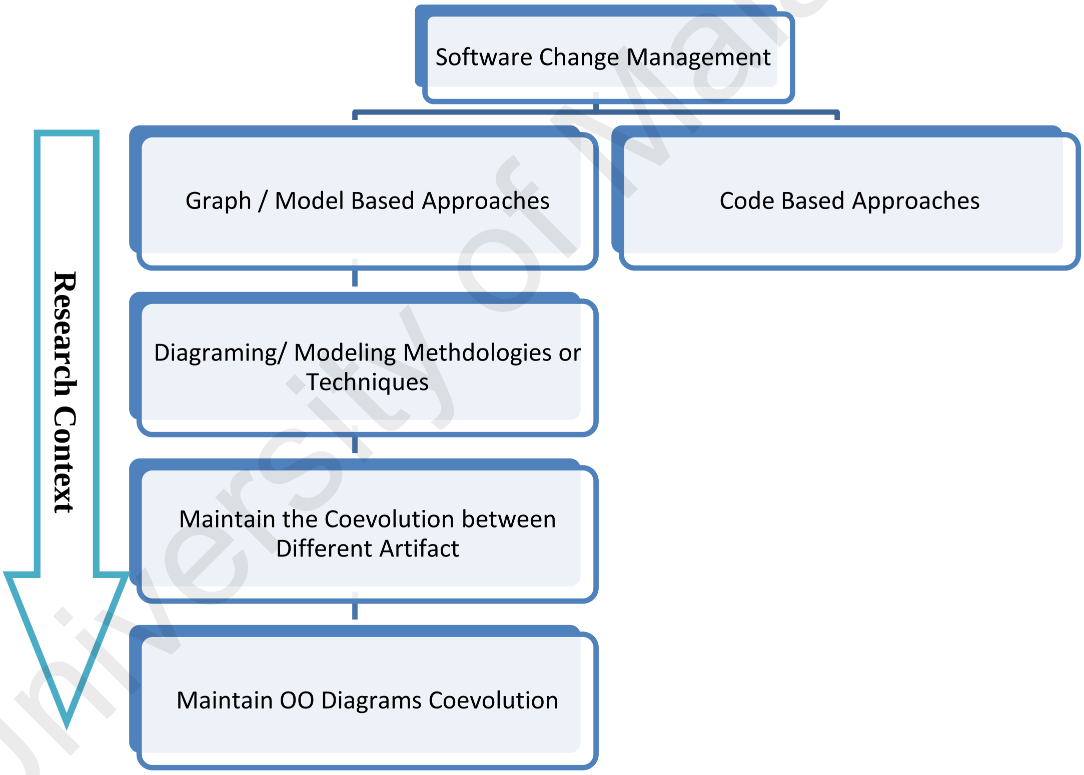

Figure 3.2: Research Context

3.2 Literature Review Phase

In this phase, various software modelling concepts and change management concepts are presented. Then, the findings from the literature review are summarized and the research direction is presented. Based on the stages discussed in Section 3.1, the literature review phase consistes of the following:

I. Studying the state of the art on consistency checking and coevolution between UML diagrams;

II. Studying the importance of design patterns in the software design process;

III. Studying the integration between UML diagrams and CPNs. This research proposes a comprehensive survey on the integration between UML diagrams and CPNs including consistency and integrity rules (Rajabi & Lee, 2009b, 2014); and

IV. Studying the state of the art on coevolution and consistency validation and verification techniques. This includes simulation techniques and consistency checking tools.

3.3 Research Design Phase

The main steps in the research design phase are shown in Figure 3.3 and are discussed in the following sections. These steps are:

I. Proposing a new structure for the integration of UML and CPNs (named Object Oriented Coloured Petri Nets (OOCPNs)) including the transformation rules to be applied between UML diagram elements and OOCPNs;

II. Proposing a set of change impact and traceability analysis templates for all types of change in most of the UML 2.3 diagrams, including rules to maintain consistency and integrity;

III. Proposing a set of coevolution patterns to model and simulate the proposed diagrams changes. This set includes the change impact and traceability analysis templates for updating UML diagrams. These patterns can help developers to build efficient models, while avoiding reinvention of already existing solutions of problems;

IV. Proposing a coevolution framework based on the proposed structure, templates, and patterns; and

V. Validating and verifying the proposed framework and checking the correctness and complexity of the proposed coevolution patterns.

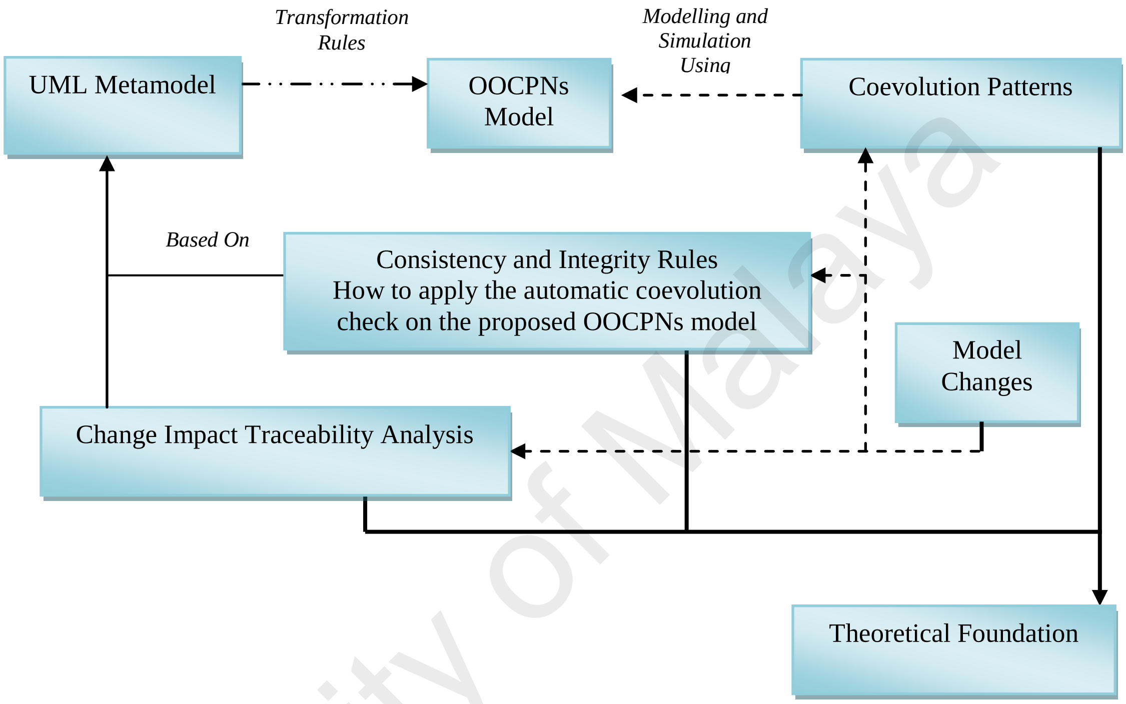

Figure 3.3: Detailed Phases of Research Methodology

3.4 Modelling and Development Phase

Based on the research justification in the previous chapters, a formal modelling language, CPNs, is used to model and simulate the proposed framework. The rationale for using CPNs stems from the fact that it provides automatic validation and verification. UML as a standard language for modelling OO software systems is a semi-formal language and does not automatically support validation and verification of the coevolution between software models. In contrast, CPNs is a formal and executable modelling language that is widely used to handle inconsistency problems among models and to automatically validate and verify the model’s dynamic behaviour. A case study is modelled in CPNs in order to apply the proposed transformation rules, change impact and traceability analysis templates, and coevolution patterns.

3.5 Analysis and Evaluation Phase

In this phase, the proposed framework is discussed and its performance is evaluated. This includes comparisons with the state of art. The main stages in this phase are:

- Providing case study models;
- Explaining the quantitative results of the research;
- Analysing and discussing the research results in comparison with those of related works. This includes a quantitative analysis of the research results. Dynamic verification of the formal method using the CPNs Tools simulation is used to verify the proposed framework. Dynamic formal analysis looks at the behaviour of the model (M Westergaard, 2007);
- Discussing the accomplishment of the research objectives; and
- Discussing the main limitations of the proposed framework.

3.6 Chapter Summary

In this chapter, the phases of the research methodology were identified and discussed. The intent of each phase was also identified. In the next chapter, the proposed coevolution framework will be discussed in detail.

CHAPTER 4: PROPOSED COEVOLUTION FRAMEWORK

In this research, a coevolution framework is proposed in order to provide a systematic and methodical approach for managing changes among UML structural, behavioural, and interaction diagrams. The proposed framework is used to check the consistency, impact, and traceability incrementally after a diagram or diagram element has been created, deleted, or modified. Additionally, the provision of a change history between two versions created from the same diagram is addressed. The coevolution and inconsistencies between UML diagrams will be detected and resolved based on a set of proposed coevolution patterns within the proposed coevolution framework.

Impact and traceability analysis is important in order to identify the parts that require retesting and to improve the overall efficiency of software change management techniques. In this research, a set of model-based change impact and traceability analysis templates is proposed for all types of change. These templates are the basis of the initiation of all update operations and are used to detect any elements affected by a change to a system modelled using UML diagrams. The nature of the change could be corrective or evolutionary. Corrective changes are implemented to correct a design error. Evolutionary changes are required due to the redesign or reconfiguration of processes. The change effect could be local if the change in one diagram does not impact on other diagrams or it could be global if it concerns relations between diagrams.

These changes are represented by consistency and integrity rules, which are discussed in Section 4.1.2. These rules are modelled using the proposed coevolution patterns. The proposed coevolution patterns are identified and categorized based on UML diagram categories and relations (structural, behavioural, and interaction diagrams).

The proposed framework is a hybrid of the transformational and formal semantic approaches. The transformational approach is required for the mutual integration of UML and CPNs modelling languages. The formal approach is used to model, simulate, and validate the proposed coevolution framework and patterns using the CPNs formal modelling tool (TGIgroup, 2013; Michael Westergaard & Verbeek, 2013).

The proposed framework, which is a type of software configuration management technique, is shown in Figure 4.1.

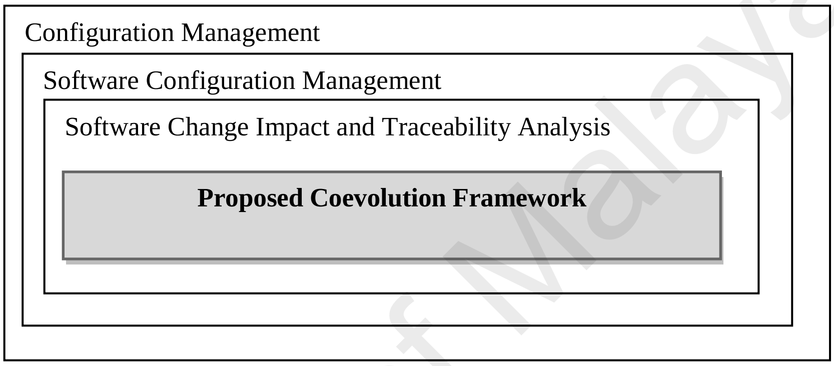

Figure 4.1: Contextual Diagram of Proposed Coevolution Framework

The main components of the proposed framework are shown in Figure 4.2 and Figure 4.3. These components are discussed in detail in the following subsections of this chapter.

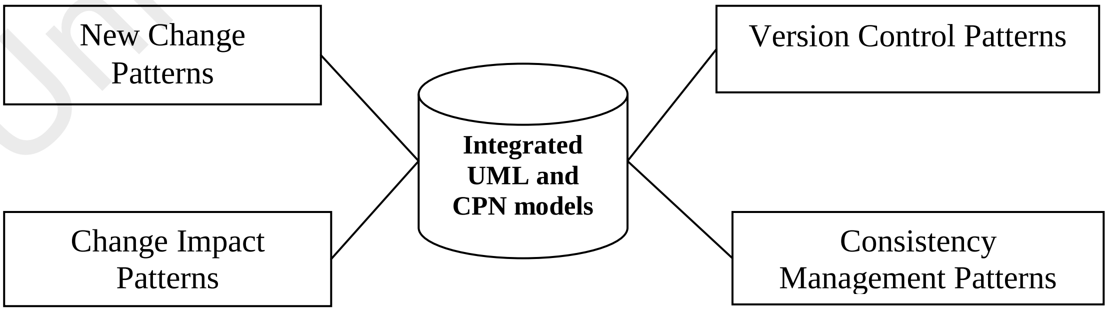

Figure 4.2: Components of Proposed Coevolution Framework

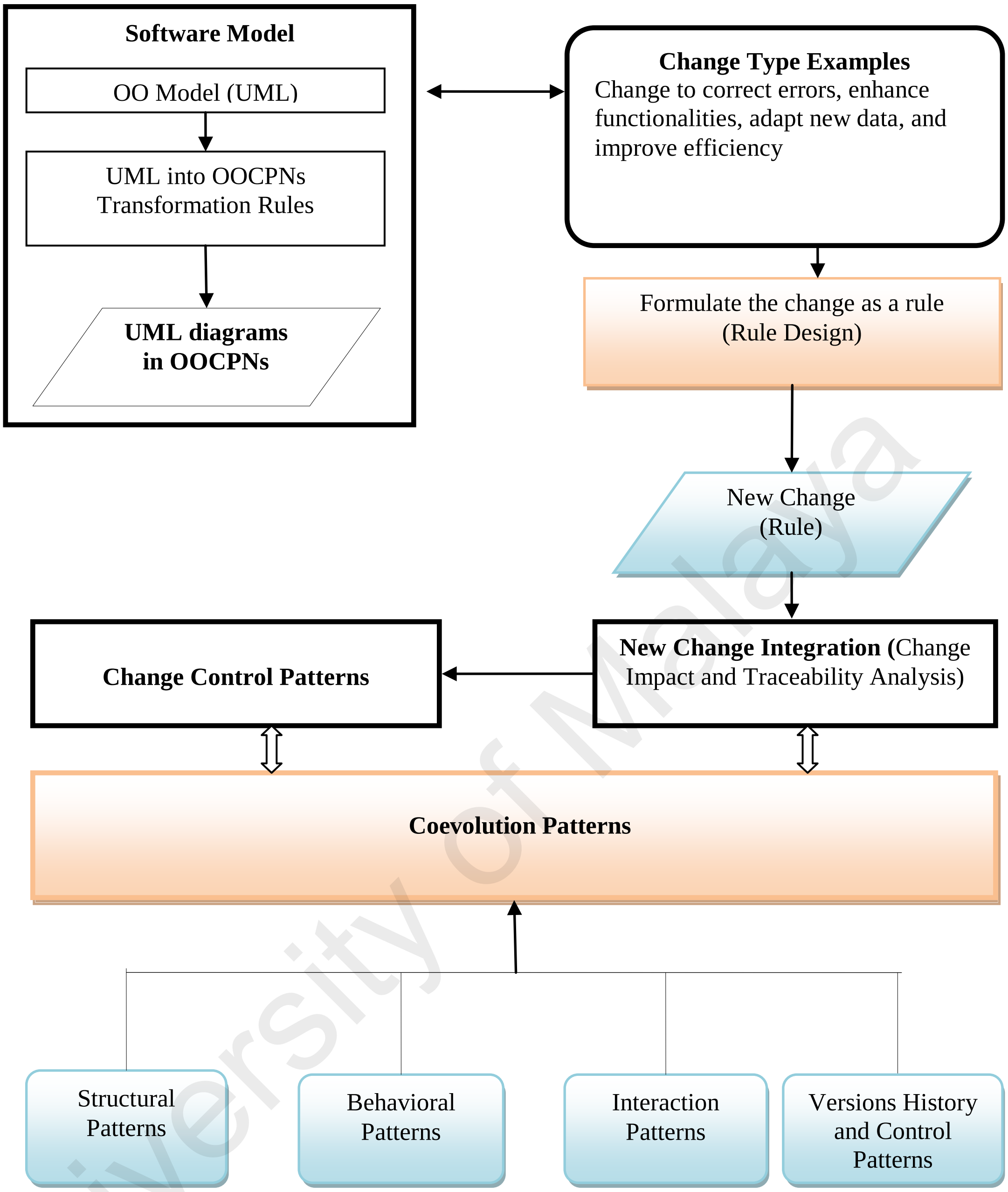

Figure 4.3: Steps of Proposed Coevolution Framework

4.1 Software Model

A complete model can be represented using UML diagrams. UML 2.3 supports a variety of diagrams which allows developers to model software systems from different perspectives using UML structural, behavioural, and interaction diagrams (OMG, 2010). UML diagrams are interrelated; some components for one diagram may be derived from other diagrams. For example, an activity diagram can be used to model an operation associated with a use case or a class diagram. Since UML diagrams can be divided into different categories, where each category focuses on a different perspective of a problem domain, one of the critical issues that needs to be addressed is the maintenance of consistency among diagrams (Shinkawa, 2006).

The patterns proposed in this research are applied to the following UML diagrams (class, object, activity, statechart, and sequence diagrams). These diagrams cover the three perspectives of UML diagrams, namely structural, behavioural, and interaction. A class diagram is useful for representing information about actors, roles, organizational units, and relevant data (Yang & Chen, 2003). Actors and data stores are objects in the object diagram. The activity diagram is concerned with the control flow and the sequence diagram is concerned with the object flow. The statechart diagram describes the process behaviour produced by states and events. The dependency between these diagrams is very high. It is crucial to transform UML diagrams into executable models that are ready for analysis, and providing an automated technique that can transform these diagrams into a mathematical model such as CPNs avoids redundancy in writing specifications.

4.1.1 Transformation of UML into CPNs

Many approaches for integrating OO modelling and PNs/CPNs have been investigated and developed. The transformation of UML diagrams into CPNs is partially supported for a subset of UML diagrams, as discussed in Rajabi and Lee (2009b). This research focuses on the transformation of UML diagrams from the structural, behavioural, and interaction perspectives. In addition, a new structure, Object Oriented Coloured Petri Nets (OOCPNs) which includes rules to maintain consistency and integrity, is proposed to support model changes. A block diagram of the transformation process is shown in Figure 4.4.

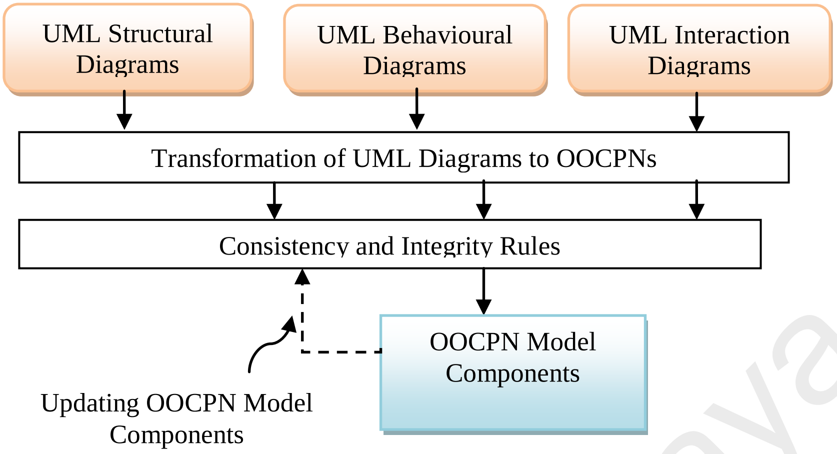

Figure 4.4: Block Diagram for Transforming UML Diagrams into OOCPNs

The components of UML structural, behavioural, and interaction diagrams are transformed into CPN elements based on the proposed transformation rules. The consistency and integrity rules are checked during the transformation process and after updating the CPN model. The proposed structure can be described formally as a tuple of

<OOCPNs structure, Relations, Rules>

The OOCPNs structure is described formally in **Definition 1**. The OOCPNs model elements are grouped together according to the relations between UML diagrams. The rules used to maintain the consistency and integrity of the transformed model are provided in **Definition 2**.

Definition 1. Proposed OOCPNs Structure:

The proposed OOCPNs structure is defined by the tuple n = (∑, *Pg*, P, Fp, T, SubT, A, N, C, G, E, M_0, R), where:

∑: is a finite set of non-empty types, called colour sets
Pg: $\{Pg_0, Pg_1 \ldots Pg_n\}$ is a set of pages, where Pg_0 is the main page
P: $\{p_1, p_2, \ldots, p_n\}$ is a finite set of places
Fp: $\{fp_1, fp_2, \ldots, fp_n\}$ is a finite set of fusion places

T: $\{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions
SubT = $\{Subt_1, \ldots, Subt_n\}$ is a finite set of substitution transitions
A: $A \subseteq T \times P \cup P \times T$ represents a set of directed arcs
N: $A \rightarrow T \times P \cup P \times T$ is a node function
C: $P \rightarrow \sum$ is a colour function
G: is a guard function
E: is an arc expression function
M_0: $P \longrightarrow C$ is the initial (coloured) marking
R: $\{r_1, \ldots, r_n\}$ is a finite set of consistency and integrity rules.

Definition 2. OOCPNs Model Relations and Rules:

The proposed transformation rules are used to transform the UML diagram elements into OOCPNs elements. The OOCPNs elements are grouped together according to the UML diagram relations as follows:

Let O be an OO software system represented by a set of UML diagram elements (E_o) where $E_o = \{E_1, E_{2,\ldots\ldots,} E_n\}$. Let $TR_o = \{TR_1, TR_{2,\ldots}TR_n\}$ be the set of transformation rules. Let $OOCPN_o = \{OOCPN_1, OOCPN_{2,\ldots} OOCPN_n\}$ be the set of equivalent OOCPNs elements of E_o. The transformation rule between $\{E_j, OOCPN_j\}$ can be defined as follows:

$$\forall \text{ Diagram element} \in E_o\text{: } E_j \underset{TRj}{\Longrightarrow} OOCPN_j \; // \; E_j \textit{ is a diagram element}$$

The OO diagrams are organized in OOCPNs as a set of {S, B, and I}, where S is the UML structural diagram elements, B is the UML behavioural diagram elements, and I is the UML interaction diagram elements. The OO diagram elements in the OOCPNs are a set of:

$\{S\,(E_1, E_{2,\ldots,} E_n), B\,(E_1, E_{2,\ldots,} E_n), I(E_1, E_{2,\ldots,} E_n)\}$

$\{CD(E_1,.., E_n), OD\,(E_1,.., E_n), AD\,(E_1,.., E_n), SCD(E_1,.., E_n), SD\,(E_1,.., E_n)\}$

The proposed transformation rules include information about the following:

- Rules to transform UML diagram elements into OOCPNs;
- Consistency and integrity rule(s) to maintain consistency and integrity during the transformation and after updating the OOCPNs model components.

4.1.2 Design of Consistency Rules

The UML structural, behavioural, and interaction diagram elements are all subject to change to accommodate new requirements. The scope of a change is determined by its impact (local or global). The types of change supported in UML diagrams are shown in Figure 4.5. The new changes are represented as rules to update diagram elements or relations incrementally. If a change to an element is based on other elements, those elements must exist. To ‘update’ means creating, deleting, or modifying diagram elements. Each update operation is represented as a pattern; examples of the proposed patterns are provided in CHAPTER 6:

Consistency and integrity rules to maintain the consistency between UML diagrams and their relations are proposed in Section 4.2. The details of the complete transformation of UML diagrams into the proposed OOCPNs structure are provided in CHAPTER 5: These rules have the structure:

```
If (set of input conditions)
        Then (set of output conditions)
Else (set of output conditions)
```

These rules are checked and applied during the change impact and traceability analysis process. Rule conditions, actions, and pre and post conditions are also considered. All consistency constraints are maintained before and after the new changes have been updated. If any one of these constraints is not satisfied then it is rejected in accordance with Rules 1 to 3 as formulated in Section 4.2. Data integrity is a critical issue and needs to be validated against certain constraints before and after applying a change. Integrity rules express constraints and define the acceptable relationships between data elements, as well as ensuring completeness. In this research, these rules are checked incrementally after each update operation, and any sequence of updates that occurs must not result in a state that violates any of the constraints. For example, the proposed rules disallow the deletion of referenced data.

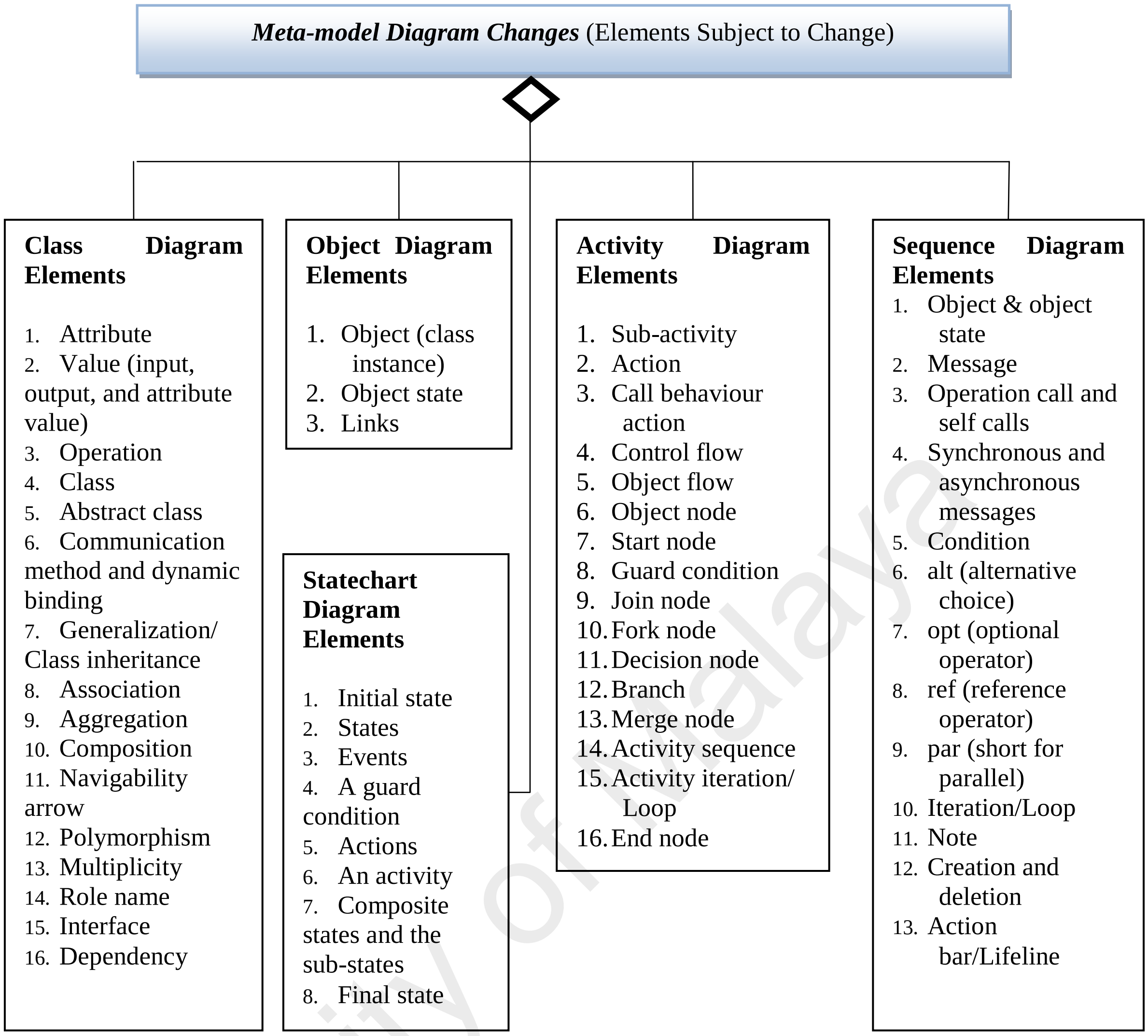

Figure 4.5: Metamodel Diagram Changes (Elements Subject to Change)

4.2 Components Affected by a Change

In the proposed patterns, the UML diagram elements affected by a change are determined based on the object dependency graph of the diagram objects and their relations. Control flow dependency and other dependencies such as inheritance, aggregation, encapsulation, polymorphism, and dynamic binding are supported by the patterns. Figure 4.6 shows a graph that represents the dependency between the UML diagrams.

Any update operation in a structural diagram will cause a change in the behavioural and interaction diagrams. Also, the behavioural and interaction diagrams are interdependent; if a change has happened in one of the behavioural diagrams, then it will affect at least one interaction diagram and vice versa. The following formal definitions (Definitions 3 to 5) are used to determine the dependencies between the UML diagram elements.

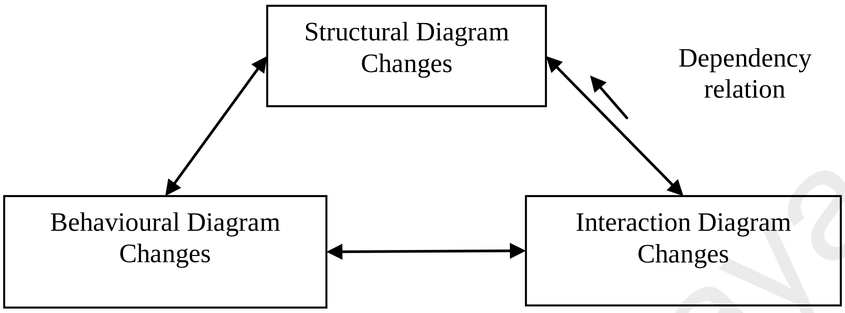

Figure 4.6: UML Diagram Dependency

Definition 3. Impact-related Elements:

Let X, Y ∈ E_o, where E_o is the set of UML diagram elements and X ≠ Y; Y is said to be an impact-related element of X, if Y is changed then X is considered changed (Briand, et al., 2003; Briand, et al., 2009). In the proposed patterns, this definition can be used to determine the impact of a change between any structural diagram's elements (**S**), behavioural diagram's elements (**B**), and interaction diagram's elements (**I**) according to the following relations:

∀ X∈ S, Y ∈ B, Z ∈ I: X is an impact-related element of Y and Z,

If (X is updated) Then (Y and Z are changed elements);

∀ X∈ S, Y ∈ B, Z ∈ I: Y is an impact-related element of X and Z,

If (Y is updated) Then (X and Z are changed elements);

∀ X∈ S, Y ∈ B, Z ∈ I: Z is an impact-related element of X and Y,

If (Z is updated) Then (X and Y are changed elements).

Definition 4. Reflexive Relation:

Given that D is the Change Impact (CI) dependency, and A is a UML diagram, the reflexive relation as defined by Lee (1998):

A D A: *A depends on itself. This means that if A is impacted, it will impact itself*

This definition describes vertical consistency, which is shown in Figure 4.7. Therefore in general, the reflexive relations are:

S D S, B D B, and **I D I**

Definition 5. Transitive Relation:

Suppose X, Y, and Z are UML diagrams, then the transitive relation as defined by Lee (1998) is:

X D Y and **Y D Z** ⇒ **X D Z** // *This means that if X impacts Y and Y impacts Z, then X impacts Z*

In the proposed patterns, examples of the transitive relations between S, B, and I are:

S D B and **B D I** ⇒ **S D I**

S D I and I D B ⇒ **S D B**

For example, a change to the class diagram will affect the activity diagram (direct impact) and a chang to the activity diagram will affect the sequence diagram (direct impact). As a result, a change to the class diagram will affect the sequence diagram (indirect impact). The change impact dependencies between the UML structural, behavioural, and interaction diagrams are defined using the relations between diagrams. The UML diagram relations are used to determine and classify all types of change in UML diagrams and the impact on other diagram elements. Horizontal, vertical, and evolutionary traceability and consistency types are supported to maintain consistency and compatibility between the UML diagrams and their versions, as shown in Figure 4.7.

The horizontal relation between the diagram elements is affected by a change and the change types can be described formally as in Definition 6. The evolutionary relation between the diagram versions can be described formally as in **Definition 7**. The change impact is determined for both direct and indirect change effects. A direct effect occurs when the change to one diagram element directly impacts the definition of another diagram element. An indirect effect occurs when the impacted diagram element in turn impacts other diagram elements.

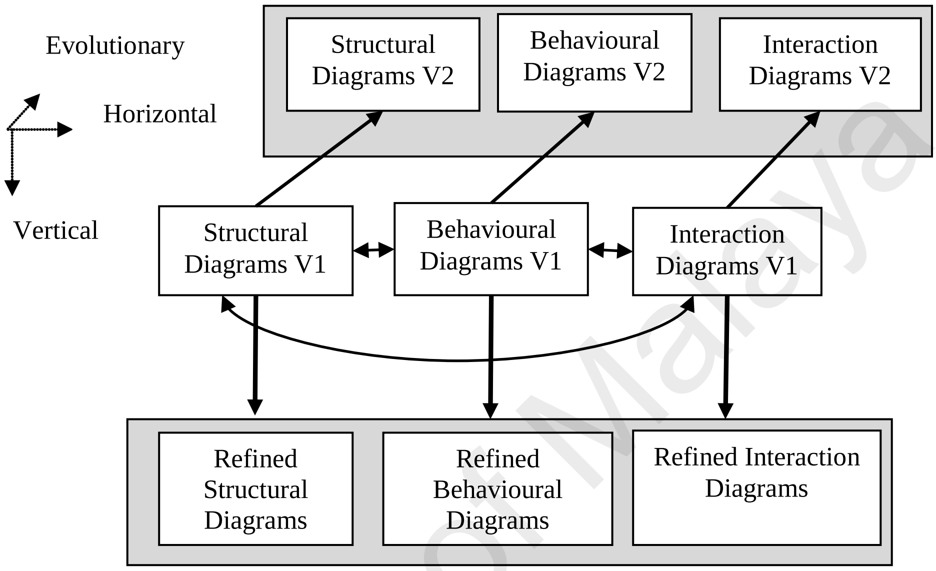

Figure 4.7: Types of Traceability and Consistency between UML Diagrams

Definition 6. Relation between UML Diagram Elements and Change Types:

Let *O* be an OO software system represented by a set of UML diagram elements (E_o), where $Eo = \{E_1, E_{2,\ldots\ldots,} E_n\}$. Let $To = \{t_1, t_{2,\ldots\ldots,} t_n\}$ be the set of change types that can be carried out on *O* such that for a given change$\{t_j, E_j\}$, we can define:

$$F_{impact}\{t_j, E_j\} \longrightarrow \{E_1, E_{2,\ldots,} E_k\} \text{ (Ajila, 1995)}$$

//where k is the number of the affected diagram elements,

where F_{impact} is the impact function and $\{E_1, E_{2,\ldots,} E_k\}$ is the set of diagram elements affected by applying change (t_j) on element (E_j). The F_{impact} can be extended to include the UML diagram categories (C): S, B, and I as in the following:

$$F_{impact}\{t_j, C_j\} \longrightarrow \{S(E_1, E_{2,\ldots,} E_k), B(E_1, E_{2,\ldots,} E_k), I(E_1, E_{2,\ldots,} E_k)\}$$

$F_{impact}\{t_j, C_j\} \longrightarrow \{CD(E_1,.., E_k), OD (E_1,.., E_k), AD (E_1,.., E_k), SCD (E_1,.., E_k), SD(E_1,.., E_k)\}$

This definition describes horizontal consistency, which is shown in Figure 4.7.

Definition 7. Relation between UML Diagram Versions:

Based on the definition of F_{impact}, the new version created from the impacted diagram elements is

$F_{impact}\{t'_j, E'_j\} \longrightarrow \{E'_1, E'_{2,\dots,} E'_{k.}\}$.

The new version from the UML diagram categories (C'): S', B', and I' is:

$\{S' (E'_1, E'_{2,\dots,} E'_k), B' (E'_1, E'_{2,\dots,} E'_k), I'(E'_1, E'_{2,\dots,} E'_k)\}$

such that: $\forall$ $E_j \in E_o$, If (E_j is changed), then (E'_j is created as new version from E_j).

The new version of the diagrams is:

$\{CD'(E'_1,.., E'_k), OD' (E'_1,.., E'_k), AD' (E'_1,.., E'_k), SCD'(E_1,.., E'_k), SD'(E_1,.., E'_k)\}$

This definition describes the relations between the UML diagram versions and the evolutionary consistency types. Definitions 1 to 5 are considered as change impact and dependency rules. The dependency between the business model's components and its impact analysis can be supported efficiently through the proposed change impact and traceability templates which include the following information for each change in the UML diagram elements (this information is the main part of the proposed patterns):

The ***Change Type*** represents the rule. It could be creating, deleting or modifying a diagram element;

The ***Change Impact*** value is 'LC' for a local change, 'GC' if the change affects other diagram elements, or 'Null' if the update operation is not allowed;

The ***Affected Diagrams (Dependency)*** is the list of the affected diagrams;

The ***Consistency and Integrity Rules*** are designed to maintain the consistency between UML diagrams and their relations. These rules are checked and applied during the change impact and traceability analysis process. The structure of the rules is provided in Section 4.1.2.

The proposed change impact and traceability analysis templates are discussed and defined formally in Section 4.3. This research proposes the following general consistency and integrity rules:

Rule 1: Deleting/modifying a referenced element

If (an update is to delete/modify a referenced element), then (deleting/modifying the referenced element is not allowed) *// A referenced element is an element defined by another diagram. For example, diagram attributes are defined by the CD.*

The change impact value will be 'Null', and the dependency value will be 'None'. The change impact and dependency value for the following examples of update operations are determined based on Rule 1:

a. Deleting the following diagram elements:

- A CD attribute, operation, class, class inheritance, association, or navigability arrow
- An object in the OD, SD.

b. Modifying the following diagram elements:

- A CD attribute name, operation name, class name, inherited class name, navigability arrow direction, polymorphic operation name, or interface element name
- An object name in the OD, SD
- A SD message name or a message attribute name.

Rule 2: Creating/deleting/modifying a non-referenced element

If (an update is to create/delete/modify a non-referenced element), then (the change impact is local).

The change impact value will be 'LC', and the dependency value will be 'None'. The change impact and dependency value for the following examples of update operations are determined based on Rule 2:

a. Creating the following diagram elements:

  - A CD value
  - An OD instance variable or variable/message data type
  - A SD note.

b. Deleting the following diagram elements:

  - A CD multiplicity range, interface, polymorphic operation or role name
  - A SD message
  - An OD instance variable
  - A SD note.

c. Modifying the following diagram elements:

  - A SCD and AD start or end node name
  - An OD instance variable or variable/message data type
  - A SD note.

Rule 3: Consistency and integrity constraints

Rule 3.1: The class attribute name and the association role name cannot have the same name (Briand, et al., 2003).

Rule 3.2: Two associations with the same name and role name are not allowed.

Rule 3.3: No private attribute or operation can be accessed by an operation of another class.

Rule 3.4: All diagram attributes/operations must be defined in the CD.

Rule 3.5: A cycle is not allowed in any directed paths.

Rule 3.6: For any update operation, the affected diagrams should also be updated.

Rule 3.7: A diagram element cannot update an attribute if the attribute changeability is not ‘changeable’.

Coevolution patterns are proposed for the changes in the UML diagram elements. These patterns can be applied to detect a direct or indirect change effect for all the diagram elements listed in Figure 4.5. These patterns also describe the change impact and traceability analysis information for UML diagram elements. This information is used in the vertical and horizontal consistency check types between UML diagrams. **Algorithm 1** given below is used to find the diagram elements affected by the change based on the objects dependency graph. Data dependency is checked a pre and post condition for each change.

Algorithm 1: Components affected by the change

```
Input: Diagram Name (N), Diagram Elements, Change Impact (CI)
Output: Diagrams Affected (Dependency)
Process:
    O: an OO software system represented by a set of UML diagram elements ($E_o$)
    D: CI dependency
    $N_o$: a set of UML diagram elements
    $N_j$: a specific element in the diagram
    S: Structural diagram elements, B: Behavioural diagram elements, I:
    Interaction diagram elements
Begin
If (CI is LC) Then
    - $N_j$ D $N_j$ // $N_j$ depends on itself. This means that if $N_j$ is impacted, it will impact
      itself.
```

```
    - $\forall N_j \in N_o$, If ($N_j$ is changed) Then ($N'_j$ is created as a new version from $N_j$)
Else //global changes
If ($N_j \in S$) Then
    - $\forall X \in S, Y \in B$, and $Z \in I$: X is an impact-related element of Y and Z,
    If (X is updated) Then (Y and Z are changed elements)
    - X', Y', and Z' are created as new versions from X, Y, and Z, respectively.
Else If ($N_j \in B$) Then
    - $\forall X \in S, Y \in B, Z \in I$: Y is an impact-related element of X and Z, If (Y is updated) Then (X and Z are changed elements)
    - X', Y', and Z' are created as new versions from X, Y, and Z, respectively.
Else (If $N_j \in I$) Then
    - $\forall X \in S, Y \in B, Z \in I$: Z is an impact-related element of X and Y,
    - If (Z is updated) Then (X and Y are changed elements)
    - X', Y', and Z' are created as new versions from X, Y, and Z, respectively.
endif endif endif
Versions update
endif
End
```

The version management technique is based on the revision version type. It stores two versions of the UML diagrams: the existing version and the newly created version.

4.3 Proposed Change Impact and Traceability Analysis Templates

In this section, the proposed change impact and traceability analysis templates are defined. The proposed templates are used to define the change type, change impact, affected diagrams, and consistency and integrity rule for each diagram element. The structural, behavioural, and interaction diagram elements together with their change types are listed in Table 4.1, Table 4.2, and Table 4.3 respectively, where the complete templates are provided in Appendix A.

The proposed impact and traceability analysis template is defined by the tuple n = (*CT*, *CI, AffectedD*, *ConstR*), where:

CT is the change type that represents the rule, which could be creating, deleting, or modifying a diagram element;

CI is the change impact value, where 'LC' denotes a local change, 'GC' denotes a change that affects the elements of other diagrams, and 'Null' is where the update operation is not allowed;

AffectedD defines affected diagrams (dependency), i.e. is a list of affected diagrams; and

ConstR defines the consistency and integrity rules to maintain the consistency between UML diagrams and their relations. These rules are checked and applied during the change impact and traceability analysis process.

Table 4.1: Structural Diagram Elements and Change Types

Diagram Element	Change Type
CD Attribute	Create an attribute
CD Operation	Create a new operation
CD Class	Create a new class
CD Generalization/Class Inheritance	Create a class inheritance
CD Association	Create an association Modify an association name
CD Aggregation	Create an aggregation
CD Composition	Create a composition
CD Navigability Arrow	Create a navigability arrow
CD Communication Method and Dynamic Binding	Create a communication method and dynamic binding
CD Polymorphism Operation	Create a polymorphic operation
CD Multiplicity	Create/Modify a multiplicity range
CD Role Name	Create/Modify a role name
CD Interface	Create an interface
CD Dependency	Create/Modify a class dependency Delete a class dependency
OD Object (Class instance)	Create a new object
OD Object States	Create/Modify a variable/message data type Create/Delete/Modify a message
PD Package	Create /Delete a package
PD Package Dependency	Create/Delete a package dependency
CoD and DD Node	Create /Delete a node
CoD and DD Component Operation	Create /Delete a new component operation
CoD and DD Dependency	Create/Delete a dependency relation
CSD Part/Port	Create/Delete a part/ port

Table 4.2: Behavioural Diagram Elements and Change Types

Diagram Element	Change Type
UCD Actor	Create an actor
UCD Communication (association)	Create/Delete communications
UCD Use case	Create a use case
UCD Extend/Include/Generalize/Use Relations	Create/Delete/Modify a use case relation
UCD Use Case Description	Create/Delete/Modify a use case description

Diagram Element	Change Type
AD Sub-Activity/SCD Activity	Create a sub-activity Delete /Modify a sub-activity
AD, UCD, and SCD, Action	Create /Delete an action Modify an action condition
AD Control Flow	Create / Delete a control flow
AD Object Flow	Create an object
AD Control Nodes (Fork, Join, Merge, and Decision)	Create/Delete/Modify a control node
AD Activity Sequence	Create/Delete/Modify an activity sequence
AD, SD, and CommD Iteration /Loop	Create/ Delete an iteration Modify an iteration decision node Modify an iteration condition
AD Call Behaviour Action	Create an AD call behaviour action
AD and SCD Start/End Nodes	Create/Delete a start or end node
SCD State	Create a state
SCD Event	Create an event
SCD, AD, and SD Guard Condition	Create/Delete/Modify a guard condition
SCD Composite State and Sub-State	The same as in SD message changes

Table 4.3: Interaction Diagram Elements and Change Types

Diagram Element	Change Type
SD Iteration /Loop	Create/ Delete an iteration Modify an iteration decision node Modify an iteration condition
SD Guard Condition	Create/Delete/Modify a guard condition
SD and CommD Object	Create an object
SD Message	Create a message
SD Operation Call	Create an operation call
SD Creation and Deletion	Create a creation and deletion
SD Synchronous and Asynchronous Messages	Create a synchronous and asynchronous message
SD Operators (alt/ opt / ref / par) Changes	Create/Delete/Modify operators
SD Action Bars/Lifelines	Create/Modify an action bar
SD and CommD Message Sequence Number	Create/Delete/Modify a message sequence number
IOD Activity or Interaction Diagram Elements	Create an activity or interaction diagram element
TD Task	Create a task
TD Task Duration	Create/Delete/Modify a task duration

4.4 Proposed Pattern Structure

The proposed UML diagram change patterns are categorized based on the UML diagram categories and relations (structural, behavioural, and interaction), as shown in Figure 4.8.

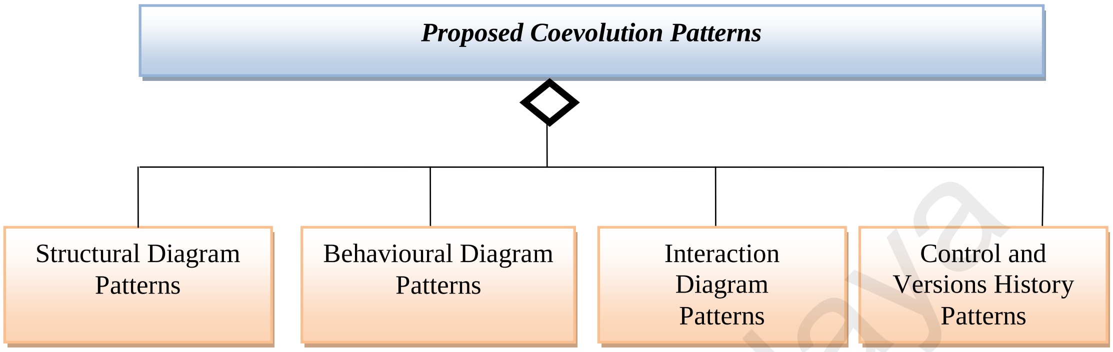

Figure 4.8: Proposed Patterns Categories

The proposed new pattern modifies Gamma , et al (Gamma , et al., 1995) and Gamma , et al (Gamma, et al., 2001) to include the change impact and traceability analysis information. The proposed pattern is defined as follows:

Pattern Name: *The identifier of a pattern that captures the main idea of what the pattern does;*

Intent: *What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?*

Motivation: *A scenario that illustrates a design problem. The scenario help to understand the more abstract description of the pattern that follows.*

Problem description: *Presents the problem addressed by the pattern;*

Solution/Diagram: *Describes possible solutions to the problem; a graphical representation of the pattern using a notation based on the proposed OOCPNs structure and CPN modelling techniques.*

***Change impact and traceability analysis**: As discussed in Section 4.2 above, this includes the following information: (Change Type, Change Impact, Affected Diagrams (Dependency), and Consistency and Integrity Rules);*

***Example:** One or more examples of the pattern found in real systems when needed. CPN places initial and final marking examples are provided.*

***Related patterns**: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?*

The proposed coevolution patterns are discussed and defined formally in CHAPTER 6: The complete lists of the proposed patterns for each diagram element are provided in Table 4.4 to Table 4.9.

Table 4.4: Proposed Class Diagram Coevolution Patterns

Diagram Element	Pattern Supported
Class	Create a class Delete a class Modify a class name Class redundancy check Class search Class with no operation or attribute Consistency check Class element redundancy check Class with no relation consistency check
Attribute	Create an attribute Delete an attribute Modify attribute name Modify attribute visibility Modify attribute property Modify attribute type Modify attribute value Attribute redundancy check Attribute search
Operation	Create an operation Delete an operation Modify operation property Modify operation type Modify operation visibility Modify operation name Operation redundancy check Operation search

Diagram Element	Pattern Supported
Generalization/Class Inheritance	Create a class inheritance Delete generalization relationship Modify generalization relationship Generalization relationship search
Association	Create an association relationship Delete an association relationship Association relationship search
Aggregation	Create an aggregation relationship Delete an aggregation relationship Aggregation relationship search
Composition	Create a composition relationship Delete a composition relationship Composition relationship search
Multiplicity	Modify association destination multiplicity Modify association source multiplicity
Role Name	Modify role name

Table 4.5: Proposed Object Diagram Coevolution Patterns

Diagram Element	Pattern Supported
Object (Class instance)	Create an object Delete an object Modify object name Search instance name Search object Exist Search instance class
Object States	Create/Delete/Modify a variable/message *These two patterns are the same as the class diagram attribute and operation patterns*
Consistency Check	Check object name Objects not created

Table 4.6: Proposed Activity Diagram Coevolution Patterns

Diagram Element	Pattern Supported
Activity	Create an activity Delete an activity Activity search
Sub-Activity	Create a sub-activity Delete /Modify a sub-activity Sub-activity search
Control Nodes (Fork, Join, Merge, and Decision)	Create/Delete/Modify a control node Fork search Join search Decision search Merge search
Object	Objects not in ADs Object search

Diagram Element	Pattern Supported
Action and Call Behaviour Action	Action search Create /Delete/Modify an action—*Lists for the activity diagram action are stored in the proposed OOCPNs structure*
Iteration /Loop	Create/ Delete/Modify an iteration—*Lists for the activity diagram loop elements (such as decision and iteration condition) are stored in the proposed OOCPNs structure* Loop Search
Guard Condition	Create/Delete/Modify a guard condition —*Lists for the activity diagram guard conditions are stored in the proposed OOCPNs structure* Guard Search
Consistency Check	ADs not created AD elements not created Modify AD name

Table 4.7: Proposed Statechart Diagram Coevolution Patterns

Diagram Element	Pattern Supported
Event	Create an event Delete /Modify an event Event search
State	Create a state
Action	Action search Create /Delete/Modify an action—Lists for the statechart diagram action are stored in the proposed OOCPNs structure
Start/End Node	Create/Delete a start or end node
Iteration /Loop	Create/ Delete/Modify an iteration—*lists for the statechart diagram loop elements (such as decision and iteration condition) are stored in the proposed OOCPNs structure* Loop Search
Guard Condition	Create/Delete/Modify a guard condition —*lists for the statechart diagram guard conditions are stored in the proposed OOCPNs structure* Guard Search
Consistency Check	SCDs not created SCD elements not created Modify SCD name

Table 4.8: Proposed Sequence Diagram Coevolution Patterns

Diagram Element	Pattern Supported
SD Object	Create an object Object search
SD Message	Message search Create a message—*list of the sequence diagram massages are stored in the proposed OOCPNs structure*
SD Iteration /Loop	Create/ Delete/Modify an iteration—*lists for the sequence diagram loop elements (such as decision and iteration condition) are stored in the proposed OOCPNs structure* Loop Search
SD Guard Condition	Create/Delete/Modify a guard condition —*lists for the sequence diagram guard conditions are stored in the proposed OOCPNs structure* Guard Search
SD Operators (alt/ opt / ref / par)	Create/Delete/Modify operators Opt search Ref search Alt search Par search
Consistency Check	SDs not created SD search SD elements not created Objects not in SDs Modify SD name

Table 4.9: Proposed Change Control Coevolution Patterns

Pattern Name	Description
Search Patterns	Find a diagram element patterns. Used to check the existing of a diagram element
Class Diagram Search Patterns	Find a class diagram element patterns
Object Diagram Search Patterns	Find an object diagram element patterns
Activity Diagram Search Patterns	Find an activity diagram element patterns
Sequence Diagram Search Patterns	Find a sequence diagram element patterns
Change History Patterns	Changes history selection Store in file Update new version

4.5 Chapter Summary

Coevolution between diagrams involves both impact analysis and change propagation. In this chapter, a coevolution framework was proposed to trace the diagram dependency and to determine the effect of the change between UML diagrams incrementally after each change operation. A set of change impact and traceability analysis templates and patterns was proposed for all types of change in the UML diagram elements. These pattern templates are the basis of the initiation of all update operations and are used to detect any elements affected by the change in the systems modelled using UML diagrams. The proposed change impact and traceability analysis templates were defined and discussed. In the next chapter, the proposed structure for the integration between UML diagrams and CPNs including the transformation rules will be defined. This integration is based on the change impact and traceability analysis templates provided in this chapter.

CHAPTER 5: TRANSFORMATION OF UML DIAGRAMS INTO CPNS

In this chapter, transformation rules to transform the structural, behavioural, and interaction elements of UML diagrams into OOCPNs are provided. The general structure for the CPN model after the transformation of UML diagrams is as follows:

Attributes and operations in the CPN model are transformed from the class diagram (CD). These attributes and operations are used by other CPN model components. Classes are organized into subpages or subnets. These subpages can be instantiated using tokens which represent the objects. Related subpages can be grouped together according to the package diagram (PD) and composite structure diagram (CSD). The behaviour and interaction of objects are described using the transformed behavioural and interaction diagrams. The statechart diagram (SCD) describes the object's behaviour by states and events. The activity diagram (AD) describes the control flow from activity to activity. The sequence diagram (SD) describes the control flow from object to object. Each activity can have a starting and finishing time to determine the sequence of activities or execution order as described in the timing diagram (TD). Communication between objects is described using SD and communication diagram (CommD). Sequence diagrams focus on the times that messages are sent. Communication diagrams focus on object roles. A communication model can be used to show the use case objects and the sequence of messages passed between them. A complete set of UML diagram elements is summarized in Figure 5.1.

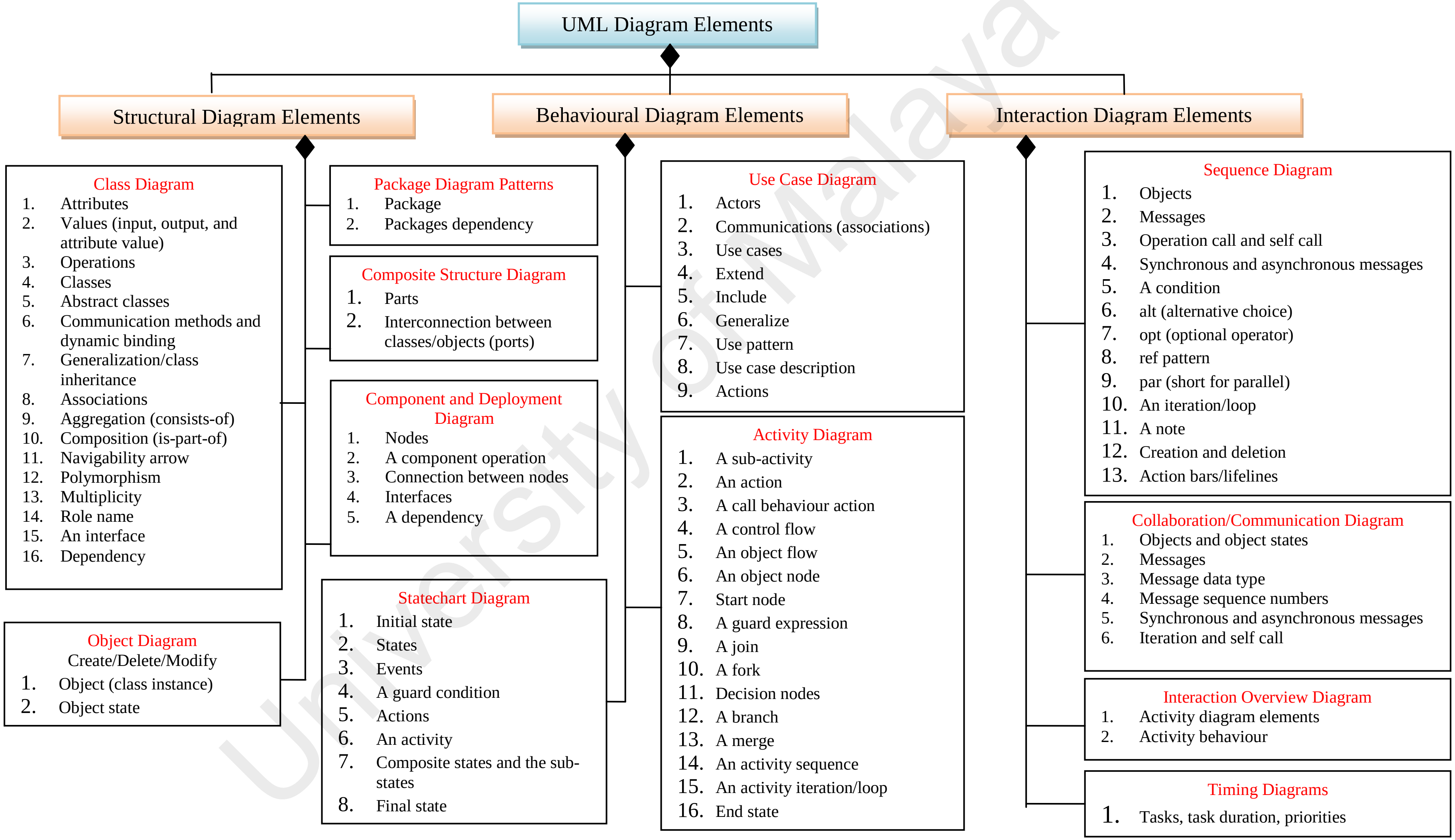

Figure 5.1: Structural, Behavioural, and Interaction in UML Diagram Elements

5.1 Class Diagram Transformation Rules

A CD is used to describe the structural and architectural composition of a system by identifying classes and their interrelations or associations. The main components for every CD are classes, associations, and multiplicities. Associations represent structural relationships between objects and describe the relationships between instances at runtime. Optional items are also provided for clarity in the CD such as navigability and roles. The role name clarifies the association nature and the navigability arrow shows the association direction. Aggregation, composition, and generalization are special kinds of associations. Multiplicity is the number of possible class instances; it can be expressed as single numbers or ranges of numbers. Examples are zero or one instance, no limit of instances, and exactly one instance. Class diagram elements are transformed into OOCPNs according to the following transformation rules:

1. CD attribute ⇒ CPN place

   Consistency and integrity rule: the same as in Template 1

2. CD attributes type ⇒ CPN colour set
3. CD values ⇒ CPN tokens *// Values: input, output, or attribute value*
4. CD value type ⇒ CPN colour set
5. CD operation ⇒ CPN subpage.

   Consistency and integrity rules: the same as in Template 2

6. CD class transformation into CPNs
   - CD class ⇒ CPN subpage
   - CD class instance ⇒ CPN substitution transition
   - CD class name and attribute ⇒ CPN place with appropriate colour type.

Example: The CD in Figure 5.2 is transformed into CPNs as shown in Figure 5.3.

Consistency and integrity rule: the same as in Template 3

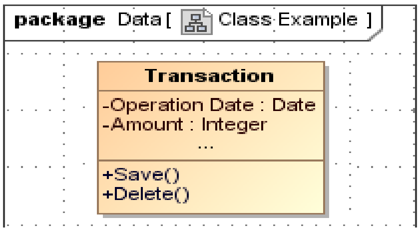

Figure 5.2: Example of Class Diagram

var Operation_Date: STRING;
var Amount: STRING;
colset Transaction = product STRING * STRING;
// Transaction class colour set is a product of the class attributes' colours

Figure 5.3: CPN ML (MetaLanguage) Description of Figure 5.2

7. CD communication method and dynamic binding transformation into CPNs

- CD synchronous request ⇒ CPN transition fusion
- CD asynchronous request ⇒ CPN fusion places

Figure 5.4 provides an example of fusion places.

Consistency and integrity rules are the same as in the SD message transformation into CPNs.

The following diagram elements are transformed into CPNs in the same way as in the CD communication method and dynamic binding:

- *SD and CommD synchronous and asynchronous messages*
- *Component Diagram (CoD) and Deployment Diagram (DD) interfaces*

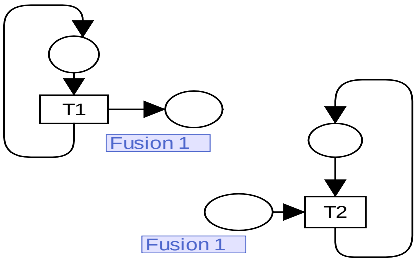

Figure 5.4: Example of Fusion Places

8. CD generalization ⇒ Hierarchical Coloured Petri Net (HCPN) by net addition (place and/or transition fusion).

   Figure 5.5 shows the transformation of generalization into CPNs. The colour set is used to model the class name, as described in Figure 5.3 for the "Transaction" class.

   Consistency and integrity rule: the same as in Template 4

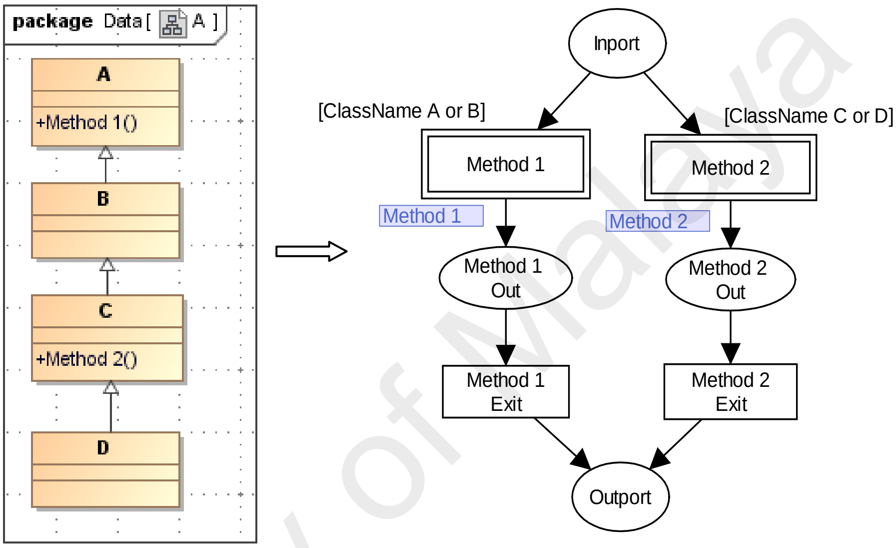

Figure 5.5: Example of CPNs for Generalization/Inheritance

9. CD associations ⇒ CPN places connected between the classes' subnets

   Consistency and integrity rule: the same as in Template 5

10. CD aggregation ⇒ HCPN by net addition (place and/or transition fusion)

   Consistency and integrity rule: the same as in Template 5

   The aggregation relation means that the target subnet needs to contain some instances of the source subnet. Communication between subnets is the same as in the CD communication method and dynamic binding. *Composition (is-part-of)* can be modelled in the same way as in aggregation, but the difference is that the target subnet needs to contain one instance of the source subnet.

11. CD navigability arrow ⇒ CPN arc

   Consistency and integrity rules: the same as in Template 6

12. CD polymorphism ⇒ HCPN by net addition (place and/or transition fusion), in addition to the net inscription as shown in Figure 5.6. An inherited attribute (polymorphism token) can hold tokens of the superclasses and subclasses. It is connected to the transition that represents the overriding operation.

    Consistency and integrity rule: the same as in Template 7

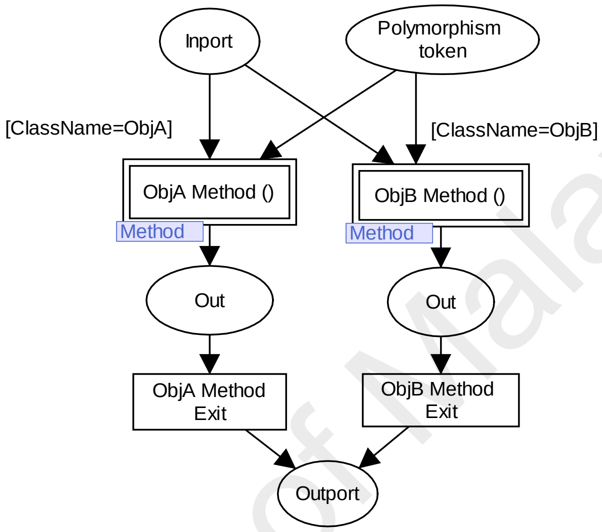

Figure 5.6: Example of CPNs for Polymorphism

13. CD multiplicity ⇒ CPN tokens and substitution transition

    Consistency and integrity rules: the same as in Template 8

14. CD role name ⇒ CPN auxiliary text

    Consistency and integrity rule: the same as in Template 9

15. CD interface ⇒ the same as in CD class transformation except that it lacks instance variables and implemented methods

    Consistency and integrity rule: the same as in Template 10

16. PD dependency ⇒ CPN arcs

    Consistency and integrity rule: the same as in Template 11

5.2 Object Diagram Transformation Rules

An object diagram (OD) consists of objects that show the instances of classes communicating by sending each other message. Attributes and behaviours/operations are the main components of the OD. Object attribute values determine the object state. Object diagram elements are transformed into OOCPNs according to the following transformation rules:

1. OD object transformation into CPNs
   - OD, SD, and CommD object (class instance)⇒ CPN tokens

     Number of tokens is equal to ($\sum Occ_i$, $i > 0$, *where* Occ_i *is the number of instances).*
   - OD object attribute ⇒ CPN token colour

     Consistency and integrity rule: the same as in Template 12
2. OD object states transformation into CPNs
   - OD instance variable ⇒ CPN place
   - OD variable type ⇒ CPN place colour
   - OD message data type ⇒ CPN product data type supported in CPNs for all the message attributes
   - OD behaviour transformation into CPNs is the same as in the CD operation transformation
   - OD communication transformation into CPNs is the same as in SD messages transformation

   Consistency and integrity rule: the same as in Template 13

5.3 Package Diagram Transformation Rules

A PD is a collection of logically related UML elements. It is used to simplify complexity in UML by grouping related classes into packages. Two packages are dependent if the change in one package could force changes in the other (Miller, 2003). Package diagram elements are transformed into OOCPNs according to the following transformation rules:

1. PD packages ⇒ HCPN by net addition (place and/or transition fusion)

   Consistency and integrity rules: the same as in Template 14

2. PD dependency ⇒ CPN arcs

   Consistency and integrity rule: the same as in Template 15

5.4 Composite Structure Diagram Transformation Rules

A CSD shows the internal structure of a class (parts) and possible collaborations (ports). It is used to explore runtime instances of interconnected instances collaborating over communication links (Ambler's, 2009). These parts must be defined in the CD or ODs. Composite structure diagram elements are transformed into OOCPNs according to the following transformation rules:

- CSD part ⇒ the same as in CD and OD element transformation
- CSD ports ⇒ CPN places

Consistency and integrity rules: the same as in Template 19

5.5 Implementation Diagrams (Component Diagrams and Deployment Diagrams)

A component is a code module. A CoD reflects the actual implementation of a system (Miller, 2003). A DD is a graph of nodes connected by communication associations. It covers the physical architecture in terms of the system hardware and software. In addition, it shows the configuration of runtime processing elements, software components, processes, and the objects that live on them (Miller, 2003).Component diagram and DD elements are transformed into OOCPNs according to the following transformation rules:

1. CoD and DD Node ⇒ subnet in HCPN, each subnet contains components and interfaces communicate together by message passing

   Consistency and integrity rules: the same as in Template 16

2. CoD and DD component operation transformation into CPNs is the same as in CD operation transformation

   Consistency and integrity rules: the same as in Template 17

3. CoD and DD dependency ⇒ CPN arc

   Consistency and integrity rule: the same as in Template 18

5.6 Use Case Diagram Transformation Rules

A use case diagram (UCD) shows actors and use cases together with their communications. It describes the functional requirements of a system in terms of actors and use cases. An actor in the UCD may be a user, an invoked application, a database, or system/device hardware. The provision of a short textual description also helps readers understand the meaning of each use case and actor. Use cases may be dependent on each other. There are many types of dependencies and relationships between use cases such as Include, Extend, Generalize, and Use. An alternative path that a use case might take if the appropriate condition holds is modelled by using the “extend” dependency. A use case that is used by other use cases is modelled by using the “include” dependency. “Use” relationships are used to show the decomposition of a use case into sub-use cases (Calderon, 2005). In the generalized interface, the child use case replaces the parent use case without interrupting the execution. This is the main difference between the “generalize” and “extend” relationships (Emadi & Shams, 2009). Use case diagram elements are transformed into OOCPNs according to the following transformation rules:

1. USD actors ⇒ CPN places

   Consistency and integrity rule: the same as in Template 20

2. UCD communications between the uses cases ⇒ CPN arcs

   Consistency and integrity rule: the same as in Template 21

3. UCD use case transformation into CPNs

- UCD use case ⇒ CPN transition
- UCD use case condition ⇒ CPN input place with transition guard

The use case can return values to the calling actors and these can also be modelled using place and transition. An example of UCD actor and use case transformation is shown in Figure 5.7.

Consistency and integrity rule: the same as in Template 22

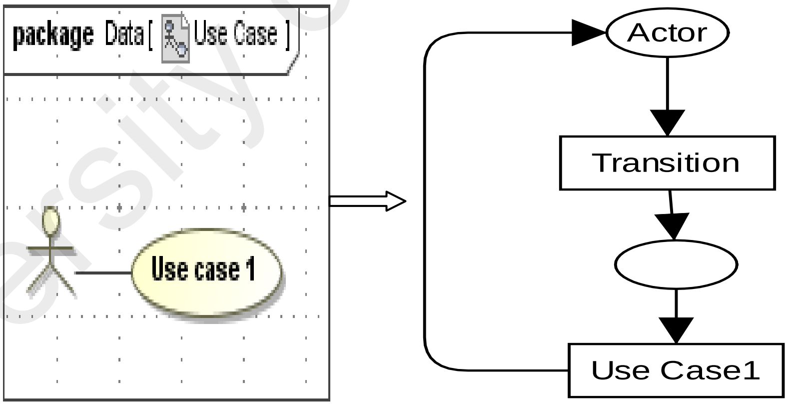

Figure 5.7: Example of Transformation of Actor and Use Case into CPNs

4. UCD use case description transformation into CPNs

- UCD action ⇒ CPN transition
- UCD action pre and post conditions ⇒ CPN transition guard function and code segment

  Consistency and integrity rule: the same as in Template 24

5. UCD extend dependency

The extend interface between two use cases is executed as follows:

If (use case B extends use case A)

Then (the execution of use case B is optional after the execution of use case A).

The extend interface between uses cases is transformed into CPNs as shown in Figure 5.8.

Consistency and integrity rule: the same as in Template 23

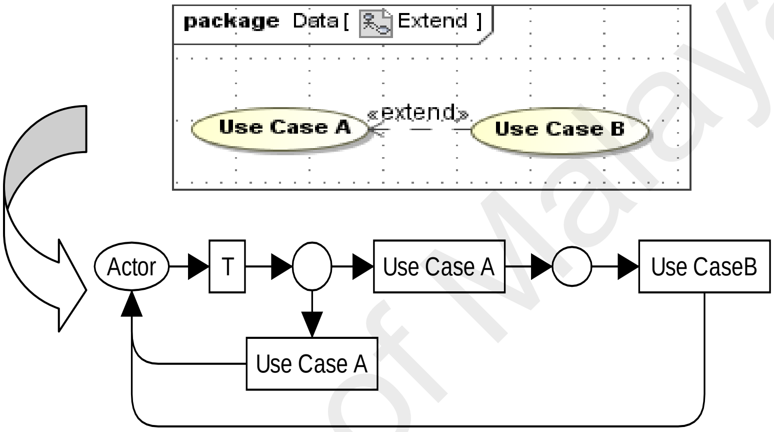

Figure 5.8: Example of transformation of extend Interface into CPNs

6. UCD include dependency

In the include interface between two use cases, the execution of the included use case is mandatory as shown in Figure 5.9.

Consistency and integrity rule: the same as in Template 23

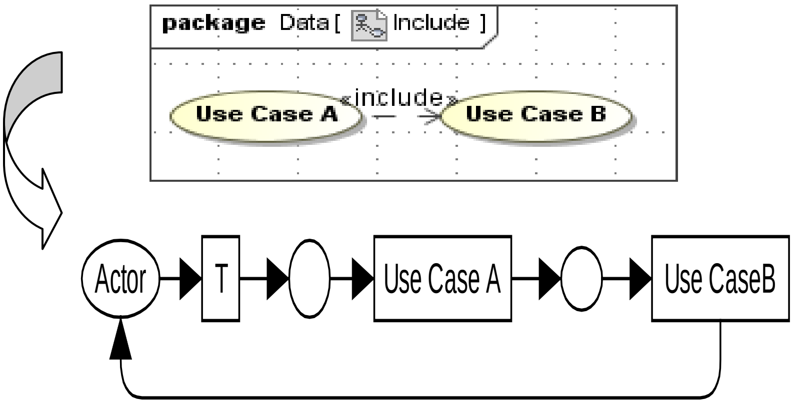

Figure 5.9: Example of Transformation of Include Interface into CPNs

7. UCD generalize dependency

   In the generalize interface, use case B can replace use case A without interrupting the execution (Emadi & Shams, 2009) as shown in Figure 5.10. This is the main difference between the generalize and extend relationships. The use relationship is transformed into substitution transitions for each use case that is decomposed into sub-use cases. Each substitution transition is modelled in the same way as in the use cases transformation into CPNs.

Consistency and integrity rule: the same as in Template 23

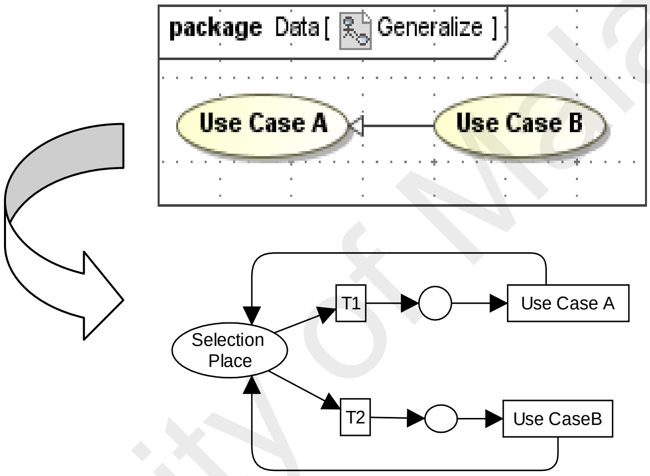

Figure 5.10: Example of Transformation of Generalize Interface into CPNs

5.7 Activity Diagram Transformation Rules

An AD is a directed graph consisting of actions and flows (Shinkawa, 2006). It focuses on the flow of activities involved in a single process and how those activities depend on one another. There are three kinds of nodes in activity models: executable/action, control, or object nodes. Other AD nodes include object swimlane, transition, branch, guard expression, and control node (Fork, Join, Merge, and Decision) (Miller, 2003). Activity diagram elements are transformed into OOCPNs according to the following transformation rules:

1. AD sub-activity/ State Chart Diagram (SCD) activity ⇒ CPN subpage

Consistency and integrity rule: the same as in Template 25

2. UCD, SCD, and AD action ⇒ CPN transition (it takes a specific input from some places and produces a specific output to other places)

   Consistency and integrity rule: the same as in Template 26

3. AD control flow ⇒ CPN places with input/output arcs

   Consistency and integrity rule: the same as in Template 27

4. AD object flow transformation into CPNs
   - AD object flow ⇒ CPN places with input/output arcs
   - AD object node ⇒ CPN place

   Consistency and integrity rule: the same as in Template 28

5. AD control nodes (Fork, Join, and Merge) transformation into CPNs
   - AD control node ⇒ CPN transition
   - AD control node input and output flow ⇒ CPN places

   AD control nodes (Fork, Join, and Merge) are modelled as a CPN transition. Each input flow and each output flow of the control node is modelled by a CPN place as shown in Figure 5.11 and Figure 5.12. The merge node and the decision node have the same notation, but in the merge node there are multiple inputs and one output (Maqbool, 2005).

   Consistency and integrity rule: the same as in Template 29

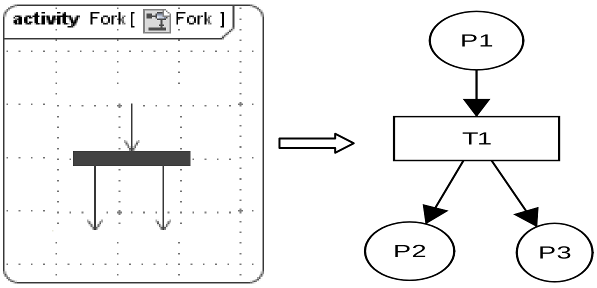

Figure 5.11: Example of Transformation of fork Node into CPNs

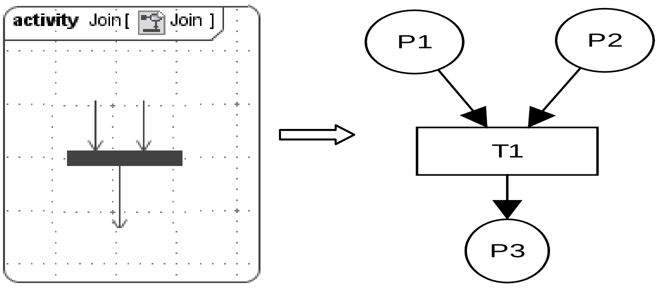

Figure 5.12: Example of Transformation of join Node into CPNs.

6. AD decision node ⇒ CPN arc inscription

The AD decision node is represented in CPNs by an arc inscription to control the passing of tokens. Tokens represent the variables' values. Each activity connected to the transition node is transformed into a CPN transition as shown in Figure 5.13. The AD branch undergoes the same transformation such that each decision node represents a branch.

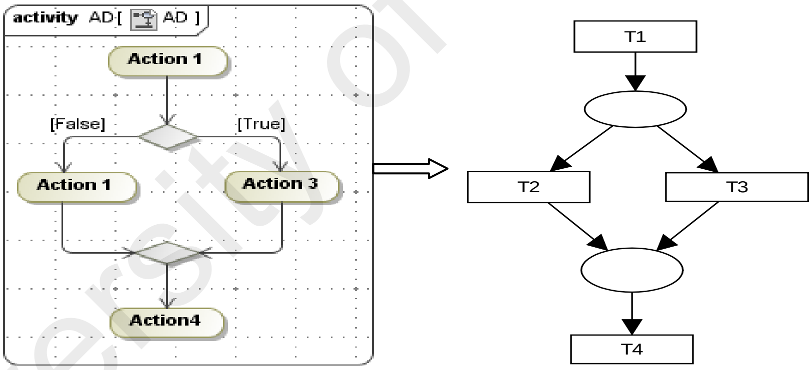

Figure 5.13: Example of Transformation of decision Node into CPNs.

7. SCD and AD start/end state transformation into CPNs
   - AD start node ⇒ CPN place without any incoming arc
   - AD end node ⇒ CPN place without any outgoing arc

Consistency and integrity rule: the same as in Template 32

8. AD activity sequence transformation into CPNs
   - AD activity sequence ⇒ CPN page including a set of interconnected activities

- AD activity ⇒ CPN transition

- AD activity input and output ⇒ CPN places

An example of the transformation of an AD start/end node and activity sequence into CPNs is shown in Figure 5.14.

Consistency and integrity rule: the same as in Template 30

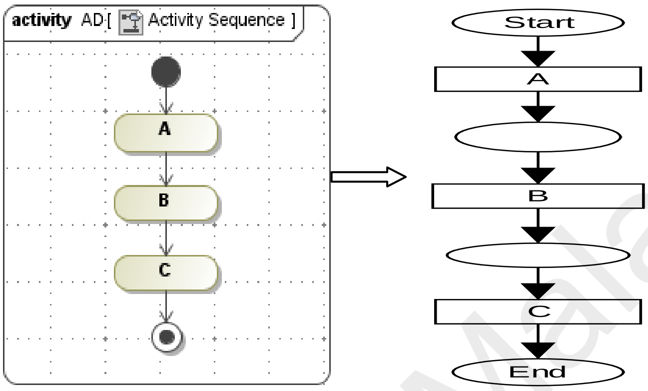

Figure 5.14: Example of Transformation of Activity Sequence and Start/End Node into CPNs

9. An example of the transformation of an AD activity iteration/loop and SD activity iteration/loop into CPNs is shown in Figure **5.15** and Figure **5.16**, respectively.

   Consistency and integrity rule: the same as in Template 31

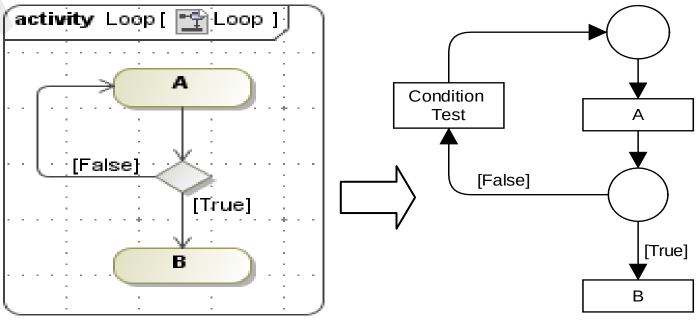

Figure 5.15: Example of Transformation of Activity Diagram Iteration/Loop into CPNs

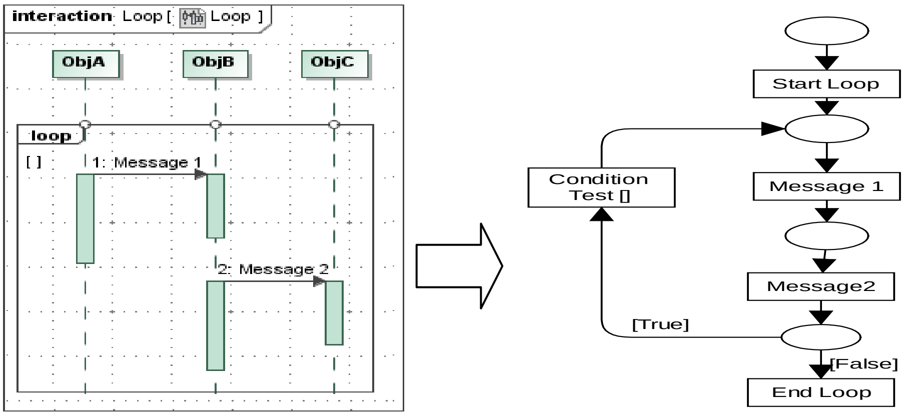

Figure 5.16: Example of Transformation of Sequence Diagram Iteration/Loop into CPNs

5.8 Statechart Diagram Transformation Rules

A SCD shows the possible states of the object and transitions (arrows from one state to another) that cause a change in states (Merseguer & Campos, 2003; Miller, 2003). A SCD contains states (simple or composite) and transitions (events or actions). Complex statecharts are those that contain composite states (Saldhana & Shatz, 2000). A state has several parts: name, entry action, exit action, internal transitions, sub-states, and deferred events. A composite state is decomposed into two or more concurrent sub-states or into mutually exclusive disjoint sub-states (Merseguer & Campos, 2003). A transition has several parts: source state, event trigger for transition firing, guard condition, and target state. Statechart diagram elements are transformed into OOCPNs according to the following transformation rules:

1. SCD state ⇒ CPN place

   //input place is for the input state and output place is for the output state.

   Consistency and integrity rule: the same as in Template 33

2. SCD event transformation into CPNs

   - SCD event ⇒ CPN transition

   - SCD event arguments ⇒ CPN token colours

   Consistency and integrity rule: the same as in Template 34

3. SCD composite state and sub-state transformation into CPNs

Composite states and sub-states are necessary when an activity involves synchronous and asynchronous sub-activities. Communications between the sub-states are described using SD and CommD message passing. Composite states and sub-states are modelled in the same way as in the SD messages transformation into CPNs and CD communication method and dynamic binding transformation into CPNs.

Consistency and integrity rule: the same as in Template 36

4. SCD note ⇒ CPN auxiliary text

SD note has the same transformation.

5.9 Sequence Diagram and Communication Diagram Transformation Rules

A SD is used to represent the life cycle of an object or the sequence of interactions between objects by message passing (how operations are carried out, what messages are sent and when) (Hu & Shatz, 2004; Khadka, 2007). Sequence diagrams are organized according to time. The vertical line represents the life cycle of an object and the horizontal line represents the interaction between objects. Objects are listed according to when they take part in the message sequence (Miller, 2003). An activation bar represents message execution duration. Iteration is represented by the asterisk on the self call. Square brackets represent the conditions. A message represents a communication between objects. Messages are classified into synchronous and asynchronous messages, based on whether the sender waits for the reply (Shinkawa, 2006). *Communication diagrams* focus on objects and their relations with the communication method, and also on object roles instead of the message times. The communication method is represented by the message flow between objects. The object roles are labelled with either class or object names or both. Sequence numbers are attached to messages to describe a certain chain of communications. Messages at the same level are sent during the same call (Miller, 2003). Sequence diagram elements are transformed into OOCPNs according to the following transformation rules:

1. SD message ⇒ CPN transition

SD messages are transformed into CPN transitions as shown in Figure 5.17. The order of transitions is according to the order of the messages in the SD. Tokens flow between places and transitions are modelled to fire the transitions (execution of messages). Places represent the objects used during message execution.

Consistency and integrity rule: the same as in <u>Template 38</u>

Transforming the following diagram elements into CPNs is the same as message transformation into CPNs:

- *CoD and DD connections*
- *AD call behaviour*
- *SD and CommD operation call*
- *SD creation and deletion*
- *CommD (messages and self call)*

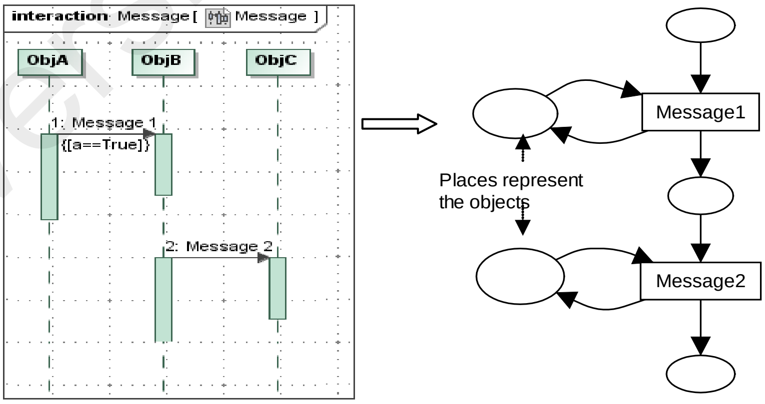

Figure 5.17: Example of Transformation of Sequence Diagram Messages into CPNs

2. SD, AD, and SCD condition ⇒ CPN place

   Consistency and integrity rule: the same as in <u>Template 35</u>

3. SD action bars/lifelines ⇒ CPN places to represent the beginning and the end of the action bar (Shinkawa, 2006)

Consistency and integrity rule: the same as in Template 41

4. SD alt

SD alt (alternative choice) is used to represent choices (nested branches). Each choice is transformed into CPNs as in messages transformation. Choices are selected for execution based on the true value of the choice guard. The branches are combined together using shared input and output places as shown in Figure 5.18.

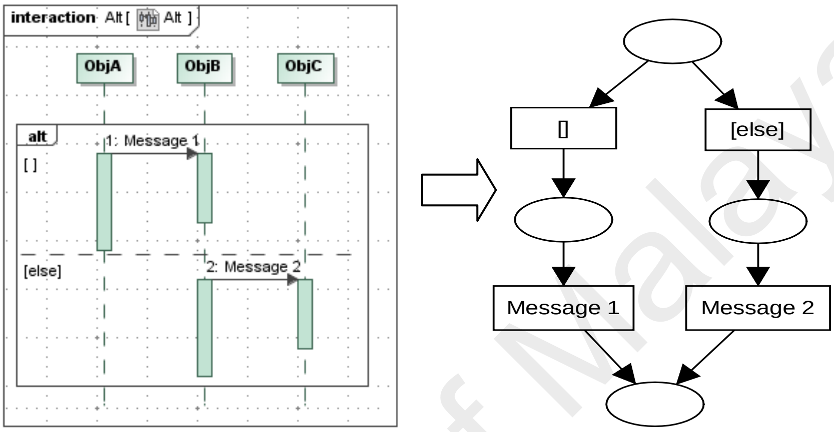

Figure 5.18: Example of Transformation of alt Operator into CPNs

Consistency and integrity rule: the same as in Template 40

5. opt (optional operator)

opt can be transformed into CPNs in the same way as in *alt* operator, because *opt* is considered as an alternative choice with only one branch whose guard is not the "else" (Ribeiro & Fernandes, 2006).

6. ref

The ref construct is transformed into a CPN substitution transition to include/reuse a SD inside another SD.

7. par (parallel)

par is used to represent number of branches that occur in parallel. Each branch is transformed into CPNs as in messages transformation, then these branches are combined together using shared input and output places and transitions as shown in Figure 5.19.

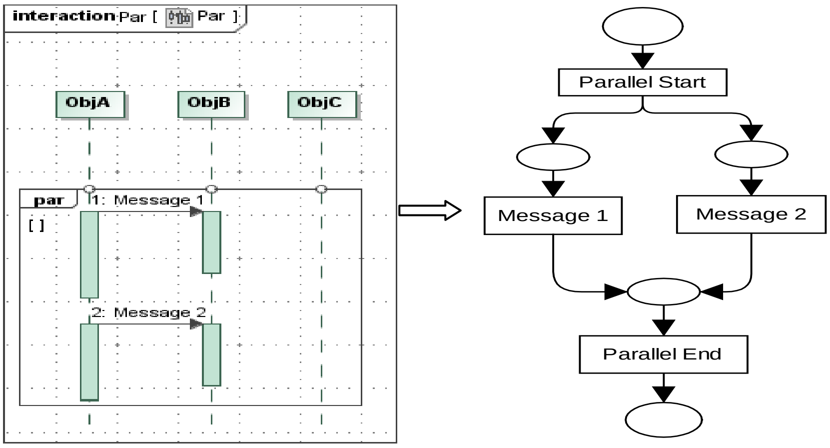

Figure 5.19: Example of Transformation of par Operator into CPNs

5.10 Interaction Overview Diagram Transformation Rule

Interaction overview diagram elements are transformed into OOCPNs according to the following transformation rule: the AD's elements are transformed as described in the AD transformation. The activity behaviour, which can be implemented using SD is transformed into a CPN subnet. The subnet is modelled as described in the SD transformation.

Consistency and integrity rule: the same as in Template 43

5.11 Timing Diagram Transformation Rules

Timing diagrams are used to explore the objects' behaviours throughout a given period of time (Ambler's, 2009). It is used for task scheduling purposes. Figure 5.20 is an example of TD modelled in CPNs. Timing diagram elements are transformed into OOCPNs according to the following transformation rules:

1. TD task ⇒ CPN transition

   Consistency and integrity rule: the same as in Template 44

2. TD duration ⇒ timed CPN token (token with time stamp)

   Consistency and integrity rule: the same as in Template 45

3. TD priority ⇒ represented by CPN ML

For example, the following ML function calculates the highest priority between two tasks:

fun higherPriority (p1, p2) =(p1>p2);

(p1 has higher priority than p2 if p1 is greater than p2 *)*

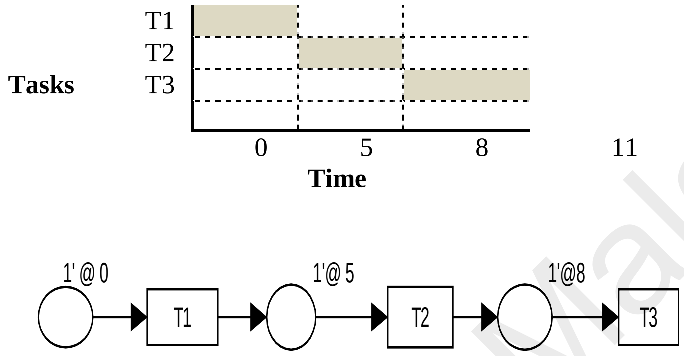

Figure 5.20: Example of Timing Diagram Modelled in CPNs

5.12 Chapter Summary

In this chapter, the transformations of the structural, behavioural, and interaction diagram UML elements into OOCPNs were provided and discussed in detail based on the proposed OOCPNs structure. In the next chapter, the proposed coevolution patterns will be defined and discussed.

CHAPTER 6: COEVOLUTION PATTERNS

Generaly, developers have focused on using patterns in software modelling as design patterns and in the workflow software management system. In this research, a new pattern design for the coevolution between UML diagrams is suggested. The proposed pattern design includes the proposed change impact and traceability analysis templates. In this work, coevolution patterns are identified and categorized based on UML diagrams categories and relations (Structural, Behavioural, and Interaction). Several issues related to the checking of the correctness of rules (changes) including the checking of data integrity and consistency, and versions history and control are discussed. Pattern simulation methodologies and results are also analyzed.

6.1 Pattern Foundation

The proposed new pattern design modifies Gamma , et al (Gamma , et al., 1995) and Gamma , et al (Gamma, et al., 2001) includes the change impact and traceability analysis information. The proposed pattern design is defined as follows:

Pattern Name: *The identifier of a pattern that captures the main idea of what the pattern does;*

Intent: *What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?*

Motivation: *A scenario that illustrates a design problem. The scenario help to understand the more abstract description of the pattern that follows.*

Problem description: *Presents the problem addressed by the pattern;*

Solution/Diagram: *Describes possible solutions to the problem; a graphical representation of the pattern using a notation based on the proposed OOCPNs structure and CPN modelling techniques.*

***Change impact and traceability analysis**: As discussed in Section 4.2 above, this includes the following information: (Change Type, Change Impact, Affected Diagrams (Dependency), and Consistency and Integrity Rules);*

***Example:** One or more examples of the pattern found in real systems when needed. CPN places initial and final marking examples are provided.*

***Related patterns**: What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?*

A summary of the proposed UML diagrams patterns and the change control patterns are provided in Figure 6.1.

6.2 Proposed Coevolution Patterns

6.2.1 Case Study Models

Case study models are modelled for the class, object, activity, statechart, and sequence diagram. These models are provided and discussed in Appendix B. All the patterns are applied based on these models. CPNs Tools simulation and monitoring toolboxes are used to validate the case study models and for monitoring and analyses. The case study models are divided in the following main sections:

Class Diagram: Figure B.3 to Figure B.13 show the class diagrams (eight classes). Additionally, the class operations and attributes are shown in each class diagram. The class diagram elements that are modelled in CPNs are *attributes, values (input, output, and attribute value), operations, classes, abstract classes, communication methods and dynamic binding, generalization/class inheritance, associations, aggregation (consists-of), composition (is-part-of), navigability arrow, polymorphism, multiplicity, role name, an interface, and dependency.*

Object Diagram: Figure B.14 and Figure B.15 show the object diagram models. Th object diagram elements that are modelled in CPNs are *object (class instance), and object state.*

Activity Diagram: Figure B.18 to Figure B.29 show the activity diagrams models. The ctivity diagram elements that are modelled in CPNs are *sub-activity, action, call behaviour action, control flow, object flow, object node, start node, guard expression, join, fork, decision nodes, branch, merge, activity sequence, activity iteration/loop, and end state.*

Sequence Diagram: Figure B.30 to Figure B.48 show the sequence diagram models. The sequence diagram elements that are modelled in CPNs are *objects, messages, operation call and self call, synchronous and asynchronous messages, condition, alt (alternative choice), opt (optional operator), ref, par, iteration/loop, note, creation and deletion, action bars/lifelines.*

Statechart Diagram: The statechart diagram elements that are modelled in CPNs are *event, state, action, start/end node, iteration/loop, and guard condition*. These elements are modelled based on the diagrams relations. Figure B.49 shows an example of a statechart diagram in CPNs.

6.2.2 Proposed Coevolution Patterns

The proposed coevolution patterns are interconnected patterns that enable incremental coevolution in a software system, which means decomposing the coevolution process into a manageable set of scenarios that can be addressed in a step-wise manner assuming that each pattern provides a solution to a given coevolution scenario. The list of proposed patterns can be found in Figure 6.1.

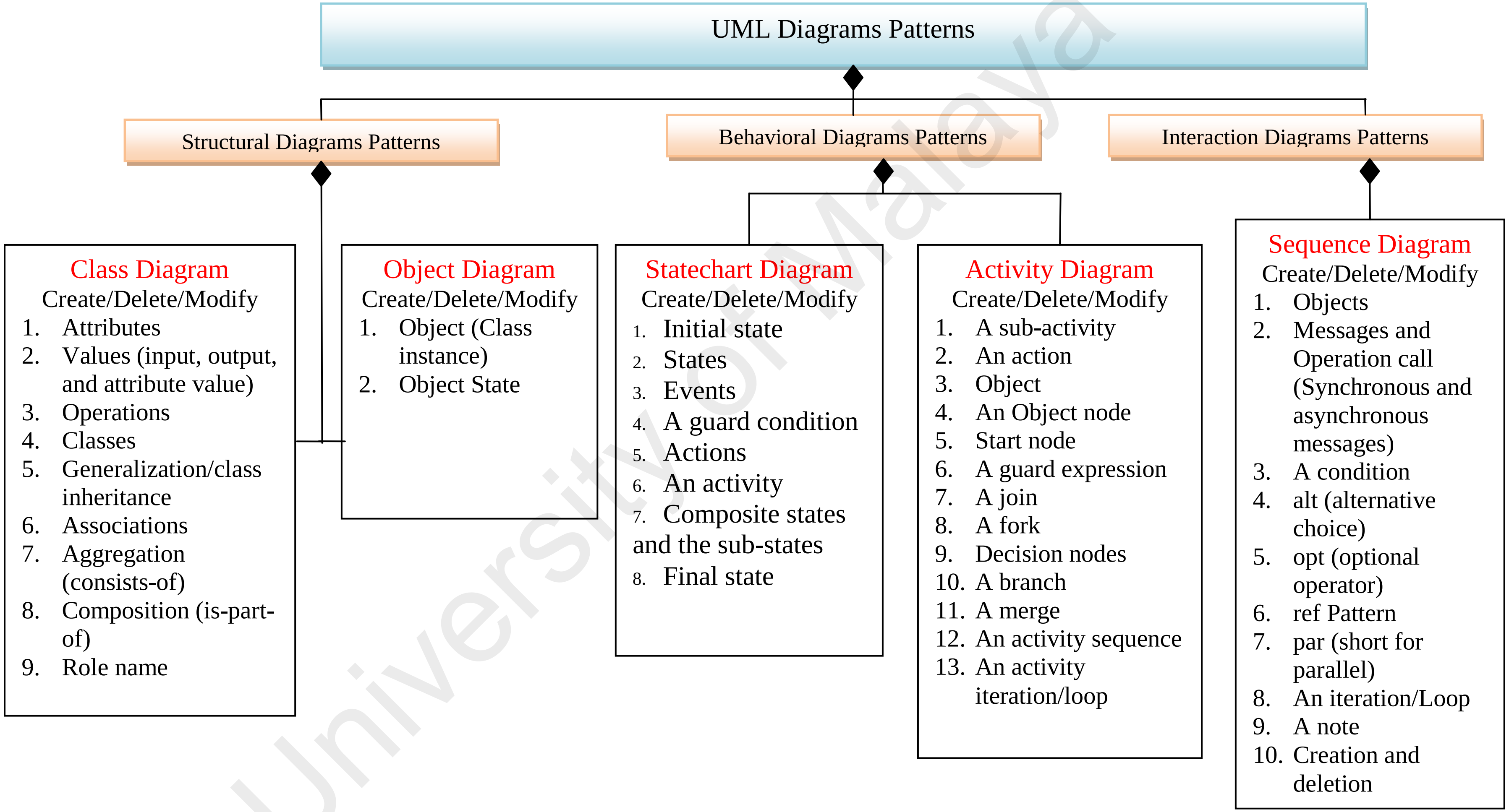

Figure 6.1: UML structural, Behavioural, and Interaction Patterns

Table 6.1 to Table 6.7 provide the main details of the proposed patterns for the class, object, activity, statechart, and sequence diagrams, respectively, grouped by the change type in addition to the change control patterns. The complete details of these patterns are provided in Appendix B and Appendix C.

Table 6.1: Proposed Class Diagram Patterns

Update Type	Patterns Group
Create an element	Create a class Create an attribute Create an operation Create a class inheritance Create an association relationship Create an aggregation relationship Create a composition relationship
Modify an element	Modify class name Modify attribute name Modify attribute visibility Modify attribute property Modify attribute type Modify attribute value Modify operation property Modify operation type Modify operation visibility Modify operation name Modify generalization relationship Modify association destination multiplicity Modify association source multiplicity Modify role name
Delete an element	Delete a class Delete an attribute Delete an operation Delete a generalization relationship Delete an association relationship Delete an aggregation relationship Delete a composition relationship
Search about an element	Class search Attribute search Operation search Generalization relationship search Association relationship search Aggregation relationship search Composition relationship search

Update Type	Patterns Group
Consistency check	Class redundancy check Class with no operation or attribute consistency check Class element redundancy check Class with no relation consistency check Attribute redundancy check Operation redundancy check

Table 6.2: Proposed Object Diagram Patterns

Diagram Element	Pattern Supported
Create an element	Create a message data type Create a variable/message *//these are* the *same as theclass diagram attribute and operation patterns*
Modify an element	Modify object name Modify a message data type Modify a variable/message
Delete an element	Delete an object Delete a variable/message
Search about an element	Search instance name Search object exist Search instance class
Consistency check	Check object name Objects not created

Table 6.3: Proposed Activity Diagram Patterns

Diagram Element	Pattern Supported
Create an element	Create an activity Create a sub-activity Create a control node Create an action Create an iteration Create a guard condition
Modify an element	Modify a sub-activity Modify a control node Modify an action Modify an iteration Modify a guard condition
Delete an element	Delete an activity Delete a sub-activity Delete a control node Delete an action Delete an iteration Delete a guard condition

Diagram Element	Pattern Supported
Search about an element	Activity search Sub-activity search Action search Fork search Join search Decision search Merge search Object search Loop search Guard search Call behaviour action
Consistency check	Objects not in ADs ADs not created AD elements not created Modify AD name

Table 6.4: Proposed Statechart Diagram Patterns

Diagram Element	Pattern Supported
Create an element	Create a start or end node Create an event Create a state Create an action Create an iteration Create a guard condition
Modify an element	Modify an event Modify an action Modify an iteration Modify a guard condition
Delete an element	Delete an event Delete a start or end node Delete an action Delete an iteration Delete a guard condition
Search about an element	Event search Action search Guard search Loop search
Consistency check	SCDs not created SCD elements not created Modify SCD name

Table 6.5: Proposed Sequence Diagram Patterns

Diagram Element	Pattern Supported
Create / Modify / Delete an element	Create an object Create a message Create/ Delete/Modify an iteration Create/Delete/Modify a guard condition Create/Delete/Modify operators
Search about an element	Object search Message search Loop search Guard search Opt search Ref search Alt search Par search
Consistency check	SDs not created SD search SD elements not created Objects not in SDs Modify SD name patterns

Table 6.6: Proposed Change Control Coevolution Patterns

Pattern Name	Description
Search Patterns	Find a diagram element patterns. Used to check the existing of a diagram element
Class Diagram Search Patterns	Find a class diagram element patterns
Object Diagram Search Patterns	Find an object diagram element patterns
Activity Diagram Search Patterns	Find an activity diagram element patterns
Sequence Diagram Search Patterns	Find a sequence diagram element patterns
Change History Patterns	Changes history selection Store in file Update new version

6.3 Patterns Simulation and Validation

In this research, the benefits of the graphical representation, simplicity, and executable nature of a CPNs model, are exploited to check the correctness of the proposed patterns and to simulate them. The correctness of the proposed patterns is checked based on the stages shown in Figure 6.2.

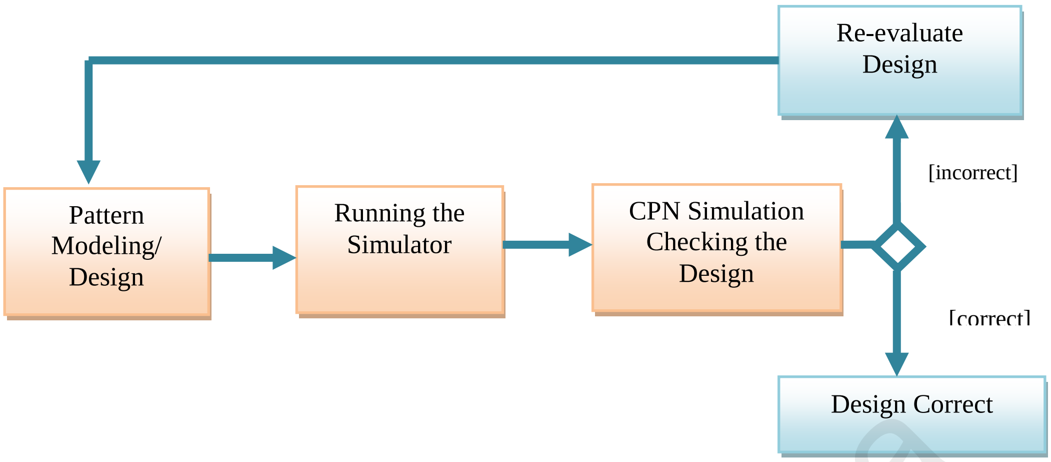

Figure 6.2: Steps for Checking Pattern Design Correctness

These stages are:

- Designing the pattern diagram;
- Running the simulation;
- The CPN simulator represents the ongoing simulation directly on the model by highlighting the enabled and occurring transitions and by showing how the markings of the individual places change.
- Some of the interactive simulation steps are controlled by some test cases to check the correctness of the model using more than one test case. Some test cases are based on automatic simulation steps.

All the designs and codes of the patterns are provided in Appendix B, Appendix C, and Appendix D. CPNs Tools provides all the means of creating the model's elements (places, transitions, arcs expressions, functions …etc). Moreover, simulation based performance analysis is supported via automatic simulation combined with data collection. The CPNs Tools toolboxes can perform a model simulation in one step or in a certain number of steps. Additionally, design verification is one of the important features in CPNs Tools. In CPNs Tools, models are verified by using different graphs. One of these graphs is a directed graph called the State Space Graph (SSG), which represents the reachable states and state changes of the model. The state explosion problem makes the verification of a large system extremely difficult.

In this research, validation and verification of the proposed patterns was done through following and tracing the simulation steps (one or a certain number of simulation steps). As shown in the patterns diagrams in Appendix C, a set of notifications and error messages is provided in these models in order to check the reachability of the nodes (places and transitions).

In the simulation steps of the proposed framework, the simulation starts with the diagram simulation (class, object, activity, statechart, and sequence). Then, the pattern models are simulated to check pattern correctness. In all steps, an initial token is provided for each of the nodes in order to trace the simulation process by transferring these tokens from the input to output places. Table 6.7 and Figure 6.3 summarize the simulation steps needed for the case study models provided in Appendix B and the proposed patterns models provided in Appendix C.

Table 6.7: Summary of Simulation Steps for Case Study Models

Diagram Element	Simulation Steps Count
Class Diagram Models	445
Object Diagram Models	246
Activity Diagram Models	503
Statechart Diagram Models	96
Sequence Diagram Models	768
Proposed Patterns Models	1301

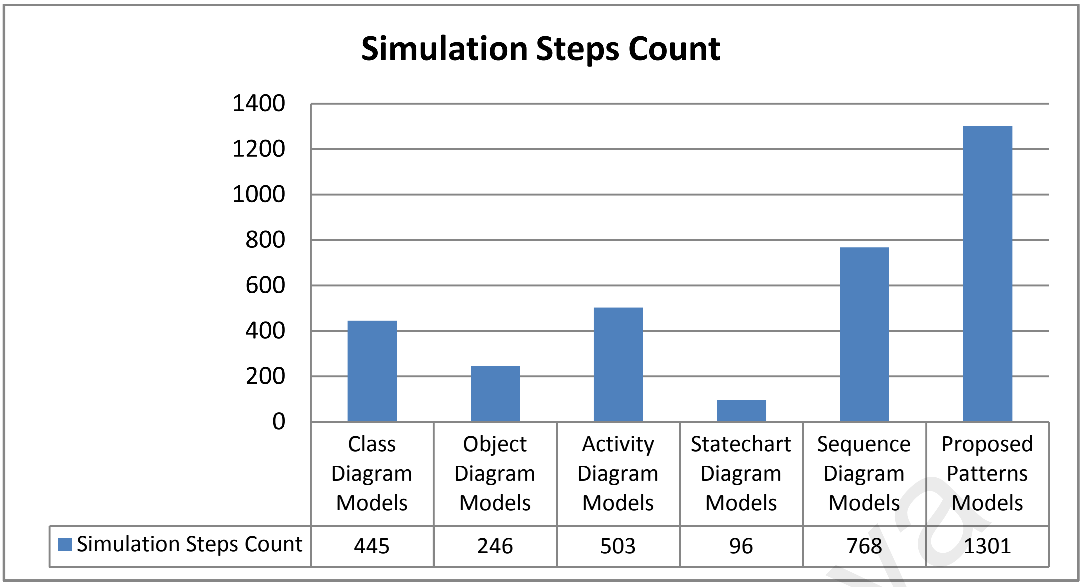

Figure 6.3: Summary of Simulation Steps for Proposed Patterns Models

In CPNs Tools, all the CPNs models can be translated into Java code using the 'Export to Java code' option provided in the Net tool box as shown in Figure 6.4.

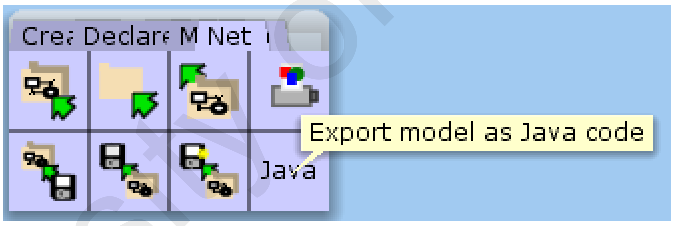

Figure 6.4: CPM Tools Toolbox for Exporting CPNs to Java Code

6.4 Chapter Summary

In this chapter, the proposed coevolution patterns foundation and relationships are identified. Additionally, the proposed patterns that are applied to trace the dependency between UML diagram elements and to determine the change effect on those UML diagrams were discussed in detail. The pattern design and simulation process was also described. In the next chapter, the proposed framework results will be analysed and discussed.

CHAPTER 7: ANALYSIS AND DISCUSSION

To accommodate changes in the software process, a framework for coevolution patterns has been proposed for determining the change effect on the various elements of UML diagrams. The proposed patterns can be applied to detect the elements affected by a change in a software system designed using UML diagrams. The framework also includes a way to control the evolution of UML diagrams by identifying and managing the model changes, ensuring the correctness and consistency of the models, identifying the impact of the changes, and determining the relationships between the model diagrams. In this chapter, the performance of the proposed framework is analysed and discussed also compared with the state-of-the-art.

7.1 Proposed OOCPNs Structure

Software models are modelled from different perspectives using UML structural, behavioural, and interaction diagrams rather than a sequence of activities. In this research a new OOCPNs structure is proposed that includes change impact and traceability analysis for UML diagrams elements.

CPNs Tools version 3.4 (Michael Westergaard & Verbeek, 2013) is used to model, simulate, and validate the transformation of UML into the proposed OOCPNs structure and patterns. This provides two main features: an executable model and an automatic consistency check. The modularity in the hierarchical structure of the proposed framework reduces interdependencies between the model components and also facilitates easy maintenance and updates without impacting the entire model. Control flow dependency ando other dependencies such as inheritance, aggregation, encapsulation, polymorphism, and dynamic binding are supported.

The proposed new OOCPNs structure supports diagrams coevolution is based on mutual integration between OO diagrams and CPNs as shown in Figure 7.1.

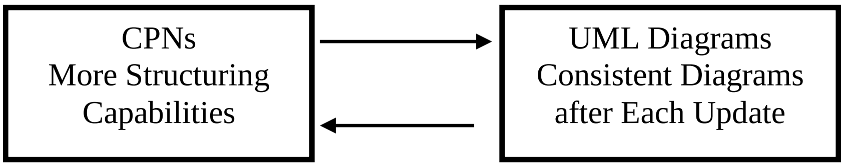

Figure 7.1: Mutual Integration between UML Models and CPNs

This mutual integration, which includes a consistency check during the transformation of UML into CPNs enhances the support for diagrams changes through building a consistent model at design time and then applying the changes to the consistent model. Table 7.1 to Table 7.3 summarize rules for transforming UML diagrams into CPNs.

Table 7.1: Rules for Transforming UML Structural Diagrams into CPNs

Template Name	Transformation into CPNs
CD Attribute Changes	CD attribute ⇒CPN place CD attributes type ⇒ CPN colour set CD values ⇒ CPN tokens *Values: input, output, or attribute value* CD value type ⇒ CPN colour set
CD Operation Changes	CD operation ⇒ CPN subpage
CD Class Changes	CD class ⇒ CPN subpage CD class instance ⇒ CPN substitution transition CD class name and attribute ⇒ CPN place with appropriate colour type.
CD Generalization/Class Inheritance Changes	CD generalization ⇒ Hierarchical Coloured Petri Net (HCPN) by net addition (place and/or transition fusion)
CD Association Changes	CD associations ⇒ CPN places connected between the classes' subnets
CD Aggregation Changes	CD aggregation and composition ⇒ HCPN by net addition (place and/or transition fusion)
CD Composition Changes	
CD Navigability Arrow Changes	CD navigability arrow ⇒ CPN arc
CD Communication Method and Dynamic Binding Changes	CD synchronous request ⇒ CPN transition fusion CD asynchronous request ⇒ CPN fusion places
CD Polymorphism Operation Changes	CD polymorphism ⇒ HCPN by net addition(place and/or transition fusion)

Template Name	Transformation to CPNs
CD Multiplicity Changes	CD multiplicity ⇒ CPN tokens and substitution transition
CD Role Name Changes	CD role name ⇒ CPN auxiliary text
CD Interface Changes	CD interface ⇒ the same as in the CD class transformation except that it lacks instance variables and implemented methods
CD Dependency Changes	CD dependency ⇒ CPN arcs
OD Object (Class Instance) Changes	OD, SD, and CommD object (class instance) ⇒ CPN tokens Number of tokens is equal to ($\sum Occ_i$, $i > 0$. *where* Occ_i *is the number of instances).* OD object attribute ⇒ CPN token colour
OD Object State Changes	OD instance variable ⇒ CPN place OD variable type ⇒ CPN place colour OD message data type ⇒ CPN product data type supported in CPNs for all the message attributes OD behaviour transformation to CPNs is the same as in the CD operation transformation OD communication transformation to CPNs is the same as in the SD message transformation
PD Package Changes	PD packages ⇒ HCPN by net addition (place and/or transition fusion)
PD Package Dependency Changes	PD dependency ⇒ CPN arcs
CoD and DD Node Changes	CoD and DD Node ⇒ subnet in HCPN, each subnet contains components and interfaces that communicate with each other by message passing
CoD and DD Component Operation Changes	CoD and DD component operation transformation to CPNs is the same as in the CD operation transformation
CoD and DD Dependency Changes	CoD & DD dependency ⇒ CPN arc **7**
CSD Part/Port Changes	CSD part ⇒ the same as in the class and object diagrams' elements transformation CSD ports ⇒ CPN places

Table 7.2: Rules for Transforming UML Behavioural Diagrams into CPNs

Template Name	Transformation to CPNs
UCD Actor Changes	USD actors ⇒ CPN places
UCD Communication (Association) Changes	UCD communications between the uses cases ⇒ CPN arcs
UCD Extend/ Include/ Use/ Generalize Relations Changes	Diagrams are provided in CHAPTER 5:.

Template Name	Transformation to CPNs
UCD Use Case Changes	UCD use case ⇒ CPN transition UCD use case condition ⇒ CPN input place with transition guard. The use case could return values to the calling actors and these are also modelled using place and transition
UCD Use Case Description Changes	UCD use case description ⇒ CPN page which includes a set of interconnected actions UCD action ⇒ CPN transition UCD action pre and post conditions ⇒ CPN transition guard function and code segment
AD Sub-Activity/SCD Activity Changes	AD sub-activity/ SCD activity ⇒ CPN subpage
AD , UCD, and SCD, Action Changes	UCD, SCD, and AD action ⇒ CPN transition (it takes a specific input from some places and produces a specific output to places)
AD Control Flow Changes	AD control flow ⇒ CPN places with input/output arcs
AD Object Flow Changes	AD object flow ⇒ CPN places with input/output arcs AD object node ⇒ CPN place
AD Control Nodes (Fork, Join, Merge, and Decision) Changes	AD control node ⇒ CPN transition AD control node input and output flow ⇒ CPN places
AD Activity Sequence Changes	AD activity sequence ⇒ CPN page including a set of interconnected activities AD activity ⇒ CPN transition AD activity input and output ⇒ CPN places
AD, SD, and CommD Iteration /Loop Changes	Diagrams are provided in CHAPTER 5:.
AD Call Behaviour Action Changes	AD Call Behaviour Action ⇒ CPN transition
AD and SCD Start/End Nodes Changes	AD start node ⇒ CPN place without any incoming arc AD end node ⇒ CPN place without any outgoing arc
SCD State Changes	SCD state ⇒ CPN place
SCD Event Changes	SCD event ⇒ CPN transition SCD event arguments ⇒ CPN token colours
SCD, AD, and SD Guard Condition Changes	SD, AD, and SCD condition ⇒ CPN place
SCD Composite State and Sub-State Changes	The same as in the SD message transformation

Table 7.3: Rules for Transforming UML Interaction into CPNs

Template Name	Transformation to CPNs
SD Iteration /Loop Changes	
SD Guard Condition Changes	SD, AD, and SCD condition ⇒ CPN place
SD and CommD Object Changes	The same as in the OD object transformation
SD Message Changes	SD message ⇒ CPN transition
SD Operation Call Changes	The same as in CD operation transformation
SD Creation and Deletion Changes	
SD Synchronous and Asynchronous Message Changes	Diagrams are provided in CHAPTER 5:.
SD Operators (alt/ opt / ref / par) Changes	Diagrams are provided in CHAPTER 5:.
SD Action Bars/Lifelines Change	SD action bars/lifelines ⇒ CPN places to represent the beginning and the end of the action bar
SD and CommD Message Sequence Number Change	Diagrams are provided in CHAPTER 5:.
IOD Activity or Interaction Diagram Elements Changes	Diagrams are provided in CHAPTER 5:.
TD Task Changes	TD task ⇒ CPN transition
TD Task Duration Changes	TD duration ⇒ timed CPN token(token with time stamp)

Figure 7.2 and Figure 7.3 summarize the number of transformation rules proposed for each diagram and for each diagrams category, respectivly.

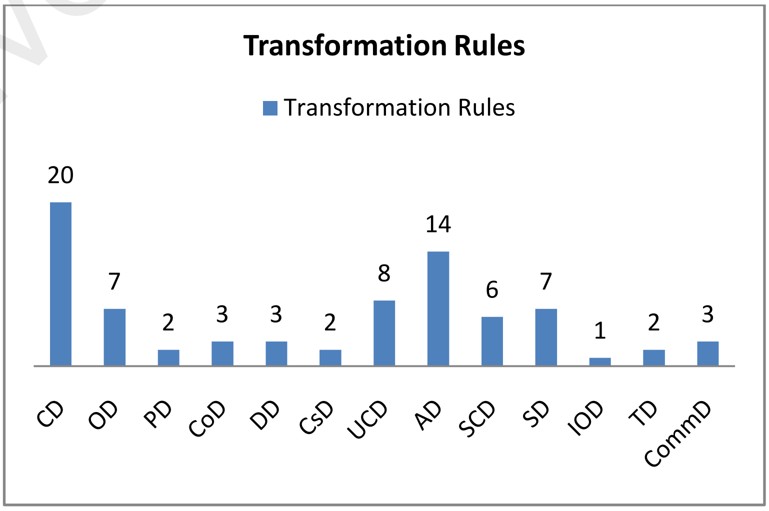

Figure 7.2: Number of Proposed Transformation Rules for Each Diagram

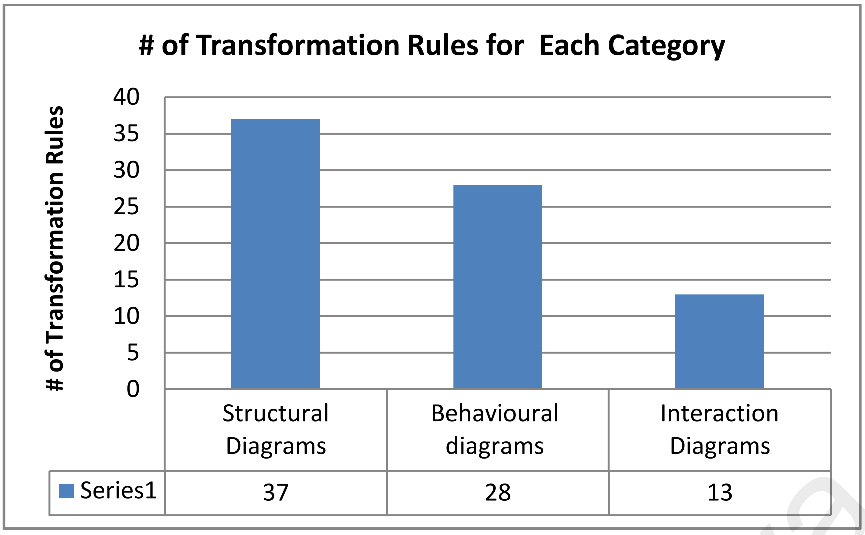

Figure 7.3: Number of Proposed Transformation Rules for Each Diagrams Category

In comparison with the approaches in (Bokhari & Poehlman, 2006; Bruckmann & Gruhn, 2008a; Wang & Wang, 2007) and with the approaches in Table 2.3, this research can be considered more comprehensive due to the greater number of UML diagrams supported in the transformation between UML diagrams and CPNs. Table 7.4 and Figure 7.4 present a comparison between the proposed OOCPNs structure and some approaches from related works in term of the number of diagrams supported in the transformation process.

Table 7.4: Comparison between the Proposed OOCPNs Structure and Selected Approaches Based on Diagrams Supported

Diagrams supported **Approach**	**Structural Diagrams**						**Behavioural Diagrams**			**Interaction Diagrams**			
	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
ArgoSPE (Gómez-Martínez & Merseguer, 2006)	√					√		√	√				
Calderon Prototype (Calderon, 2005)	√						√				√		
Baresi (2002)	√								√		√		
Barros and Gomes (2004) Wang (2007)	√												
Bokhari and Poehlman (2006)									√				
van der Aalst (2002)								√	√	√			
Guerra and de Lara (2003)	√								√	√			
Abstract Node (2006)	√									√			
Shin et al. (2003), Barros and Jorgensen (2005)	√						√				√		
AMABULO(Bruckmann & Gruhn, 2008a; Brückmann & Gruhn, 2008b)	√							√	√				
Graph Transformation (Y. Zhao, et al., 2004)									√				
Emadi and Shams (2008, 2009)					√		√			√			
Maqbool (2005), Liles (2008), Bouabana-Tebibel (2007), Garrido and Gea (2002)								√					
Proposed transformation in this research	√	√	√	√	√	√	√	√	√	√	√	√	√

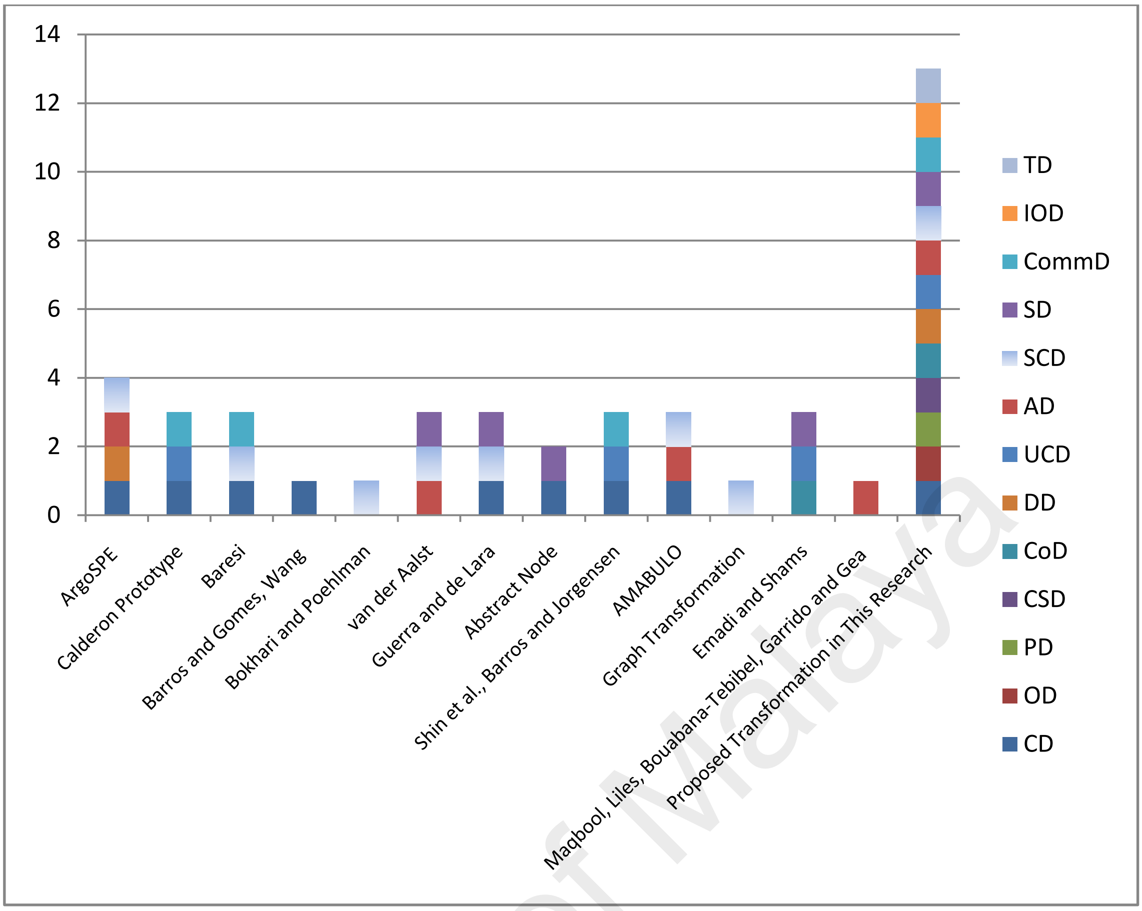

Figure 7.4: Comparison between the Proposed OOCPNs Structure and Selected Approaches Based on Diagrams Supported

7.2 Change Impact and Traceability Analysis Templates

This research proposed 45 templates as explained in Appendix A. Some of these templates are shared between multiple diagrams based on the relations between diagrams. Table 7.5 to Table 7.7 summarize the proposed change impact and traceability analysis templates.

Table 7.5: Change Impact and Traceability Analysis Templates for UML Structural Diagrams

Template Name	Change Type
CD Attribute Changes	Create an attribute
CD Operation Changes	Create a new operation
CD Class Changes	Create a new class
CD Generalization/Class Inheritance Changes	Create a class inheritance
CD Association Changes	Create an association Modify an association name

Template Name	Change Type
CD Aggregation Changes	Create an aggregation
CD Composition Changes	Create a composition
CD Navigability Arrow Changes	Create a navigability arrow
CD Communication Method and Dynamic Binding Changes	Create a communication method and dynamic binding
CD Polymorphism Operation Changes	Create a polymorphic operation
CD Multiplicity Changes	Create/Modify a multiplicity range
CD Role Name Changes	Create/Modify a role name
CD Interface Changes	Create an interface
CD Dependency Changes	Create/Modify classes dependency Delete a class dependency
OD Object (Class Instance) Changes	Create a new object
OD Object State Changes	Create/Modify a variable/message data type Create/Delete/Modify a message
PD Package Changes	Create /Delete a package
PD Package Dependency Changes	Create/Delete a package dependency
CoD and DD Node Changes	Create /Delete a node
CoD and DD Component Operation Changes	Create /Delete a new component operation
CoD and DD Dependency Changes	Create/Delete a dependency relation
CSD Part/Port Changes	Create/Delete a part/ port

Table 7.6: Change Impact and Traceability Analysis Templates for UML Behavioural Diagrams

Template Name	Change Type
UCD Actor Changes	Create an actor
UCD Communication (Association) Changes	Create/Delete communications
UCD Use Case Changes	Create a use case
UCD Extend/Include/Generalize/Use Relations Changes	Create/Delete/Modify a use case relation
UCD Use Case Description Changes	Create/Delete/Modify a use case description
AD Sub-Activity/SCD Activity Changes	Create a sub-activity Delete /Modify a sub-activity
AD , UCD, and SCD, Action Changes	Create /Delete an action Modify an action condition
AD Control Flow Changes	Create / Delete a control flow
AD Object Flow Changes	Create an object
AD Control Nodes (Fork, Join, Merge, and Decision) Changes	Create/Delete/Modify a control node
AD Activity Sequence Changes	Create/Delete/Modify an activity sequence
AD, SD, and CommD Iteration /Loop Changes	Create/ Delete an iteration Modify an iteration decision node Modify an iteration condition
AD Call Behaviour Action Changes	Create an AD call behaviour action

Template Name	Change Type
AD and SCD Start/End Node Changes	Create/Delete a start or end node
SCD State Changes	Create a state
SCD Event Changes	Create an event
SCD, AD, and SD Guard Condition Changes	Create/Delete/Modify a guard condition
SCD Composite State and Sub-State Changes	The same as in the SD message changes

Table 7.7: Change Impact and Traceability Analysis Templates for UML Interaction Diagrams

Template Name	Change Type
SD Iteration /Loop Changes	Create/ Delete an iteration Modify an iteration decision node Modify an iteration condition
SD Guard Condition Changes	Create/Delete/Modify a guard condition
SD and CommD Object Changes	Create an object
SD Message Changes	Create a message
SD Operation Call Changes	Create an operation call
SD Creation and Deletion Changes	Create a creation and deletion
SD Synchronous and Asynchronous Message Changes	Create a synchronous and asynchronous message
SD Operators (alt/ opt / ref / par) Changes	Create/Delete/Modify operators
SD Action Bars/Lifelines Changes	Create/Modify an action bar
SD and CommD Message Sequence Number Changes	Create/Delete/Modify a message sequence number
IOD Activity or Interaction Diagram Elements Changes	Create an activity or interaction diagram element
TD Task Changes	Create a task
TD Task Duration Changes	Create/Delete/Modify a task duration

Figure 7.5 shows the distribution of these templates over the UML diagrams categories. In total, 22 templates are proposed for structural diagrams, 18 templates are proposed for behavioural diagrams, and 13 templates are proposed for interaction diagrams. Some of these templates are shared by more than one diagram based on the relations between the diagrams. For example, the same template is proposed for the activity diagram and sequence diagram iteration /loop changes.

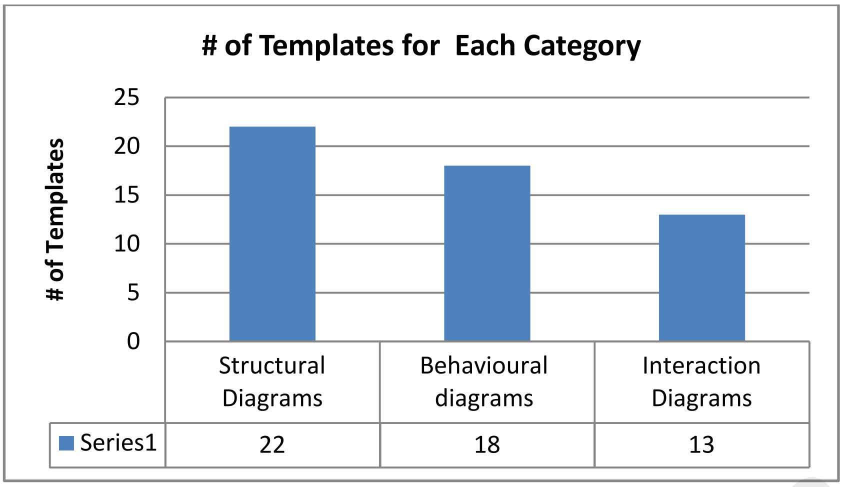

Figure 7.5: Number of Proposed Templates for each Diagrams Category

Figure 7.6 show the number of proposed templates for each structural diagram.

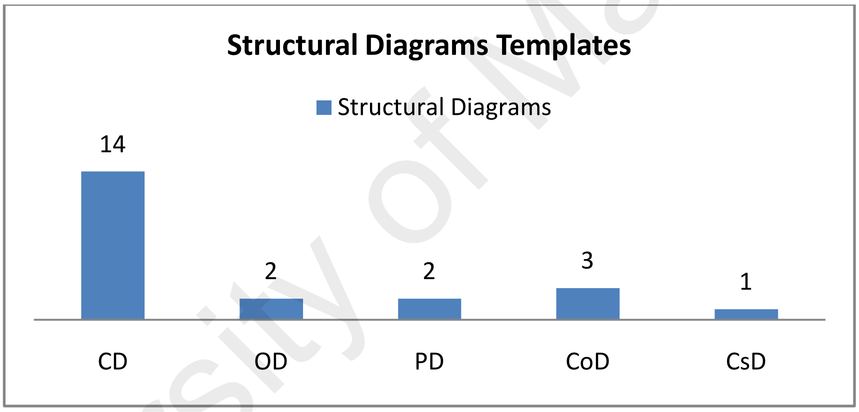

Figure 7.6: Number of Proposed Templates for Each Structural Diagram

Figure 7.7 show the numbersof proposed templates for each structural diagram.

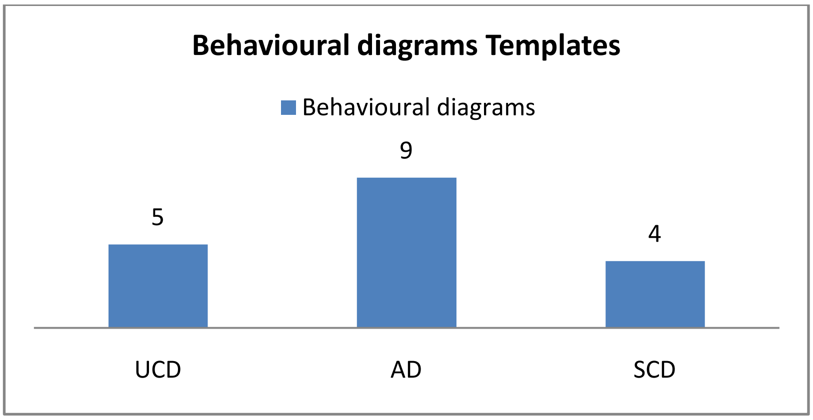

Figure 7.7: Number of Proposed Templates for Each Behavioural Diagram

Figure 7.8 show the number of proposed templates for each structural diagram.

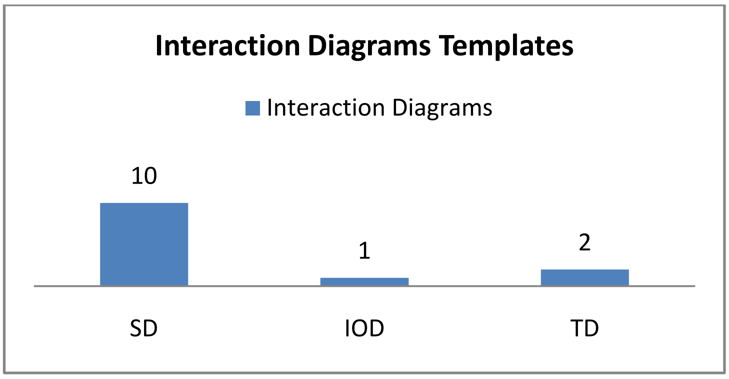

Figure 7.8: Numbersof Proposed Templates for Each Interaction Diagram

7.2.1 Evaluation Metrics

In this research, quantification of the change impact is based on two metrics: the set of diagrams/ diagrams elements affected by the change and the change levels.

A. Metrics for Change Level

An algorithm has been proposed to determine the change impact and the dependency between the elements the UML diagrams. Corrective and evolutionary changes are supported. Figure 7.9 shows the hierarchy of the change levels.

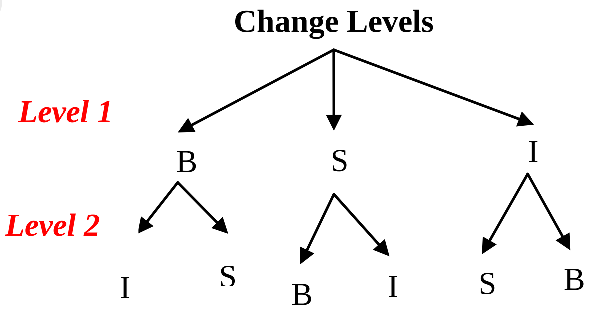

Figure 7.9: Hierarchy of Change Levels (Traceability Distance)

The change level is used to determine the distance between the changed element and the impacted elements. The change distance is calculated according to the following rule:

If (the change in S, B, or I is local)

Then (change distance is 1)

Else (change distance is 2). *//the number of affected diagrams (n) by the change is $n \geq 1$.*

B. Metrics for Affected Diagrams and Elements

This metric is related to the set of diagrams or diagram elements affected by a change. It is also referred to as the cost of the change. The higher the impact on the diagrams and elements, the more severe the change. As shown in the figures and tables provided in this section, the results show that the relation between the class diagram and other models is strong. This explains the large number of change impact templates and patterns proposed for the class diagram.

The dependency between UML diagrams has also been defined formally in Definitions 1 to 5. The change impact on the diagrams' elements can be defined based on the dependency relations; some examples of these relations are given below:

- ∃ *e(diagram element)* ∈ CD: If (*e* is changed) Then (all diagrams are affected)

  Classes, attributes, and operations in the class diagram are used or invoked in all UML diagrams.
- ∃ *e* ∈ OD: If (e is changed) Then (all diagrams are affected except the CD)

  Objects are used in the structural, behavioural, and interaction diagrams
- ∃ *e* ∈ CoD : If (*e* is changed) Then (DD is affected)

  CoD and DD are dependent on each other; a change in one of them will affect the other.
- ∃ *e* ∈ DD: If (*e* is changed) Then (CoD is affected)
- ∃ *e* ∈ UCD: If (*e* is changed) Then (AD, SCD, SD, CommD, TD, and IOD are affected)

  The dynamic behaviour of the UCD is described using the AD, SCD, SD, and CommD. The flow of control in the AD is from activity to activity. The flow of control in the SD and CommD is from object to object. TD and IOD are affected indirectly by the change in the UCD because their elements are derived from the AD and interaction diagrams.

- ∃ *e* ∈ AD: If (*e* is changed) Then (UCD, SCD, SD, CommD, IOD, and TD are affected)

An AD represents the internal behaviour of the CD, UCD, and SCD. The IOD and TD elements are derived from the AD elements, in addition to interaction elements added in the IOD. The AD shows how those activities depend on one another.

- ∃ *e* ∈ SCD: If (*e* is changed) Then (UCD,AD, SD, CommD, TD, and IOD are affected)

The dynamic behaviour of the SCD is described using the AD, SD, and CommD. TD and IOD are affected indirectly by the SCD changes because their elements are derived from the AD and interaction diagrams.

- ∃ *e* ∈ SD: If (*e* is changed) Then (UCD, AD, SCD, CommD, and IOD are affected)
- ∃ *e* ∈ CommD: If (*e* is changed) Then (UCD, AD, SCD, SD, and IOD are affected)
- ∃ *e* ∈ PD, CSD, IOD, and TD: If (*e* is changed) Then (no diagrams are affected)

Table 7.8 illustrates the change effect on the diagrams and diagrams elements based on the proposed templates. The table also shows the elements that are shared between diagrams. These shared elements represent the relationships between the templates. The same thing will be applied to the patterns relations. *Note that in the table, the symbol '√' means the diagram is affected and in some cases examples of the affected elements are provided.*

Table 7.8: The Change Effect on Diagrams Elements Based on the Proposed Templates

Template # / Diagram	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
Template 1. CD Attribute Changes	√	√ Object States	√	√	√	√	√	√ Object States	√ variables	√ Object States	√ Object States	√	√
Template 2. CD Operation Changes	√	√ Object States	√	√	√ component operation	√ component operation	√ Use case	√ Activities and Sub Activities, Actions	√ Events	√ Sequence diagrams states, Messages	√ Messages	√	√
Template 3. CD Class Changes	√	√ Object Instance	√	√	√	√	√	√ Object Instance	√	√ Object Instance	√ Object Instance	√	√
Template 4. CD Generalization/Class Inheritance Changes	√	√	√	√	√	√	√	√	√	√	√	√	√
Template 5. CD Association Changes	√	√	√	√	√	√	√	√ Seq. of Activities, cntrl node, call behaviour	√	√ operators	√	√	√
Template 6. CD Navigability Arrow Changes	√	√ Object Flow	√	√	√	√	√	√ Object and Control Flow	√	√ Object Flow	√	√	√
Template 7. CD Polymorphism Operation Changes	√	√	√	√	√	√	√	√	√	√	√	√	√
Template 8. CD Multiplicity Changes	√												
Template 9. CD Role Name Changes	√												
Template 10. CD Interface Changes	√												

Template # / Diagram	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
Template 11. CD Dependency Changes	√		√ dependency		√ dependency	√ dependency							
Template 12. OD Object (Class instance) Changes		√ Object Instances	√	√	√	√	√	√ Object Instances	√	√ Object Instances	√ Object Instances	√	√
Template 13. OD Object States Changes		√ Object States						√ Object States		√ Object States	√ Object States		
Template 14. PD Package Changes	√												
Template 15. PD Package Dependency Changes	√												
Template 16. CoD and DD Node Changes	√				√ Node	√ Node							
Template 17. CoD and DD Component Operation Changes					√ component operation	√ component operation							
Template 18. CoD and DD Dependency Changes					√ dependency relation	√ dependency relation							
Template 19. CSD Part/Port Changes				√									
Template 20. UCD Actor Changes							√	√	√	√	√	√	√
Template 21. UCD Communication (association) Changes							√			√ Objects links	√ Objects links		
Template 22. UCD Use case Changes							√	√	√	√	√	√	√
Template 23. UCD Extend/Include/Generalize/Use Relations Changes							√	√ Seq. of Activities, cntrl node, call behaviour		√ operators	√	√	

Template # / Diagram	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
Template 24. UCD Use Case Description Changes							√	√ sequence of Activities					√
Template 25. AD Sub-Activity/SCD Activity Changes							√	√ Activity/ Sub-Activity	√ Event	√ Operators	√	√	√
Template 26. UCD, SCD, and AD Action Changes							√ Action	√ Action	√ Action	√ Operators	√	√	
Template 27. AD Control Flow Changes							√	√ Object Flow		√ Object Flow	√ Object Flow	√	
Template 28. AD Object Flow Changes								√ Object/ control Flow		√ Object Flow	√ Object Flow		
Template 29. AD Control Nodes (Fork, Join, Merge, and Decision) Changes							√ relationships	√ Control Nodes		√ operators	√	√	
Template 30. AD Activity Sequence Changes							√ description	√	√	√	√	√	√
Template 31. AD, SD, and CommD Iteration /Loop Changes							√	√ Loop/ branches	√ Loop/ branches	√ Loop/ branches	√ Loop/ branches	√	
Template 32. AD and SCD Start/End Nodes Changes								√	√				
Template 33. SCD State Changes							√	√	√	√	√	√	
Template 34. SCD Event Changes							√	√	√	√	√	√	
Template 35. SCD, AD, and SD Guard Condition Changes							√ Guard	√ Guard	√ Guard	√ Guard	√ Guard	√ Guard	

Template # / Diagram	CD	OD	PD	CSD	CoD	DD	UCD	AD	SCD	SD	CommD	IOD	TD
Template 36. SCD Composite State and Sub-State Changes									√	√ message passing	√ message passing		
Template 37. SD and CommD Object Changes							√	√	√	√	√	√	
Template 38. SD Message Changes	√ Comm method dynamic binding						√	√	√	√ synchronous asynchronous messages	√ synchronous asynchronous messages	√	
Template 39. SD Synchronous and Asynchronous Messages Changes	√ Comm method dynamic binding				√ interface	√ interface			√ events	√ synchronous asynchronous messages	√ synchronous asynchronous messages		
Template 40. SD Operators (alt/ opt / ref / par) Changes							√ Description, relationships	√ Cntrl node branches		√	√	√	
Template 41. SD Action Bars/Lifelines Changes								√ Activity sequence		√		√	√
Template 42. CommD Message Sequence Number Changes											√		
Template 43. IOD Activity or Interaction Diagram Elements Changes												√	
Template 44. TD Task Changes													√
Template 45. TD Task Duration Changes													√

Information about the number of diagrams affected by updating each UML diagram and the number of update operations supported is provided in Table 7.9, Figure 7.10, and Figure 7.11. Self, direct, and indirect dependencies are considered. Further information about the dependency between diagrams (change effect between diagrams) is provided in Figure 7.12.

Table 7.9: Statistics in the Effect of Updating UML Diagram Elements

Diagram Name	No of Affected Diagrams	No of Update Operations
Class Diagram (CD)	13	41
Object Diagram (OD)	12	9
Package Diagram (PD)	1	5
Component Diagram (CoD) and Deployment Diagram (DD)	2	13
Composite Structure Diagram (CSD)	1	6
Use Case Diagram (UCD)	7	12
Activity Diagram (AD)	7	17
Statechart Diagram (SCD)	7	18
Sequence Diagram (SD)	6	23
Communication Diagram (CommD)	6	17
Interaction Overview Diagram (IOD)	1	3
Timing Diagram (TD)	1	5

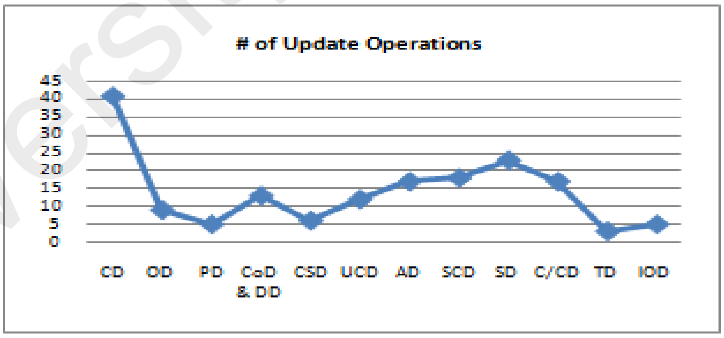

Figure 7.10: Number of Update Operations Supported by Each UML Diagram

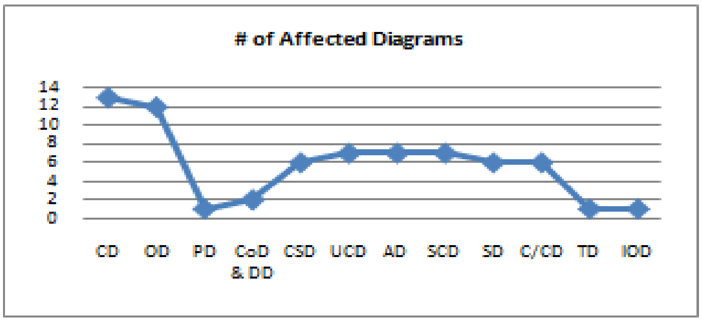

Figure 7.11: Number of Diagrams Affected by Updating UML Diagram

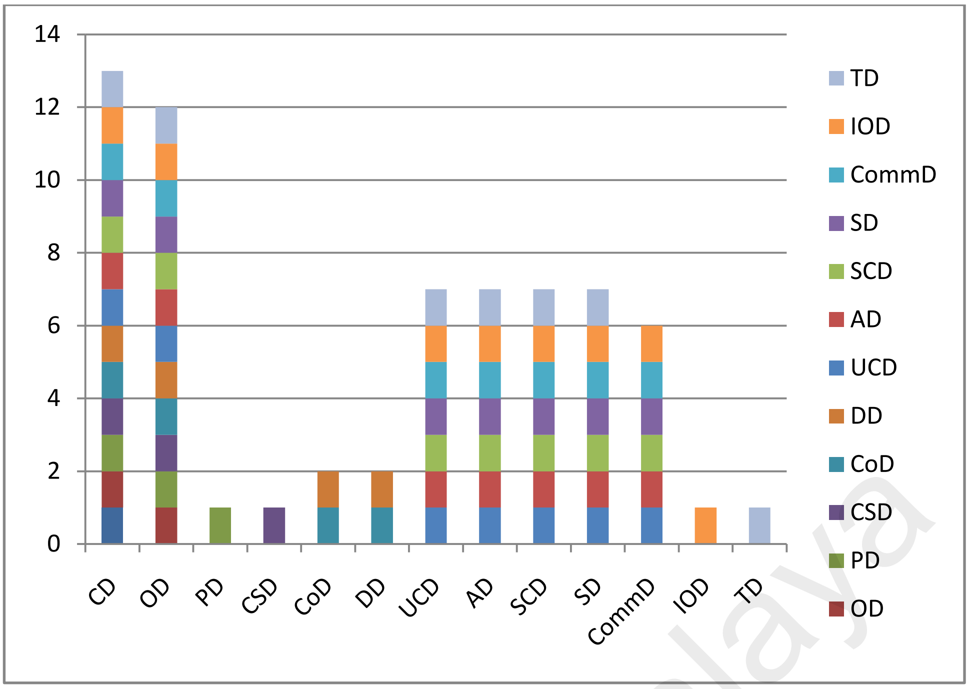

Figure 7.12: Diagrams Dependency/Change Effect

In comparison with the approaches mentioned in Table 2.3 such as (Gongzheng & Guangquan, 2010; Shinkawa, 2006), the change impact and traceability analysis rules are supported in the transformation between UML diagrams and CPNs and for most of the UML diagrams. Additionally, It is not check only the consistency between two versions from the same diagram as proposed by (Van Der Straeten, et al., 2003) and other approaches. In the state of the art approaches (Al-Khiaty & Ahmed, 2016; Lehnert & Riebisch, 2013; Li, et al., 2012; X. Sun, et al., 2010), some metrics (such as precision, recall, and F-measure) are used in determining the average time needed to detect the inconsistencies between diagrams and the effectiveness of the change impact. These metrics are used for code-based change impact analysis techniques.

7.3 Coevolution Patterns

In this research, coevolution patterns are proposed as a way to determine and classify the types of changes in UML diagrams and their impact on other diagrams. The consistency between diagrams is checked according to the consistency and integrity rules provided in each pattern. Vertical, horizontal, and evolutionary consistency types are checked. The proposed patterns trace the dependency and determine the effect of a change in the UML diagrams elements incrementally; the patterns are used to check the consistency, impact, and traceability after creating, deleting, or modifying any diagram element by applying the same idea of syntax checking incrementally to CPNs. A comparison of two versions derived from the same diagram is supported. The proposed patterns were discussed in detail in CHAPTER 6: The main elements in the proposed patterns are:

Pattern Name: short description of the problem, its solution, and consequences;

Problem: when to apply the pattern (problem, and context);

Solution: generalized solution to the problem; and

Related Patterns: show the dependency between diagrams elements.

In this research, the challenge was to propose a set of empirically gathered patterns in OOCPNs in pattern format. The main goal was to find a way to utilize OOCPNs patterns as a source of sound solutions for problems that may appear during modelling. In order to help developers in selecting a suitable pattern, this research classifies the patterns and analyses the relationships between the patterns to enable easy navigation through the patterns.

This research proposes 84 patterns to support changes in the diagrams elements as shown in Figure 7.13 and Figure 7.14. The new proposed pattern design modifies Gamma , et al (Gamma , et al., 1995) and Gamma , et al (Gamma, et al., 2001) to include the change impact and traceability analysis information.

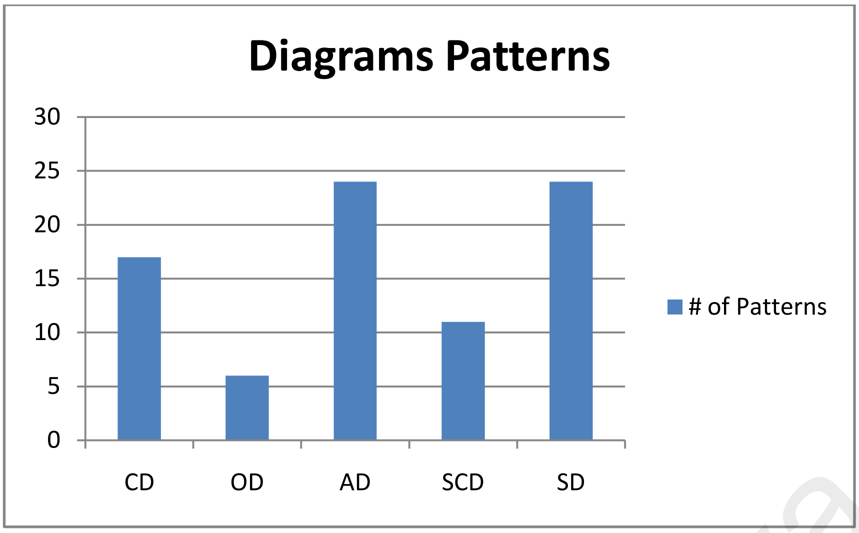

Figure 7.13: Diagrams Patterns

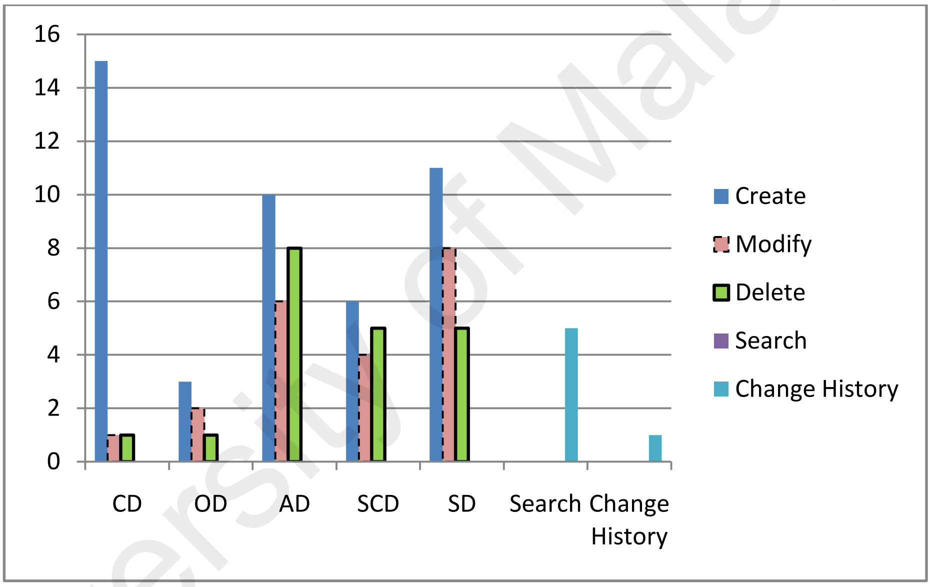

Figure 7.14: Number of Proposed Patterns

Table 7.10 summarizes the change effect on the diagrams and diagrams elements based on the proposed templates and patterns. Additionally, this table provides the relationships and intersections between the proposed templates and patterns. Where there is an intersection, this means that the pattern or template is shared between diagrams and it can be applied to the intersecting diagrams elements. Note that in the table, the *symbol '√' means the diagram is affected and in some cases examples of the affected elements are provided.*

Table 7.10: The Patterns, Templates, and Diagrams affected Relationships

Patterns Provided	**Template**	**Affected Diagrams and Elements**				
		CD	**OD**	**AD**	**SD**	**SCD**
Pattern 1. Attribute Redundancy Check Pattern Pattern 4. Class with No Operation or Attribute Consistency Check Pattern Pattern 5. Class Element Redundancy Check Pattern Pattern 8. CD Attribute Search Pattern Pattern 50. CD Create New Attribute Patterns Pattern 57. CD Delete Attribute Patterns Pattern 64. CD Modify Attribute Name Patterns Pattern 65. CD Modify Attribute Visibility Patterns Pattern 66. CD Modify Attribute Property Patterns Pattern 67. CD Modify Attribute Type Patterns Pattern 68. CD Modify Attribute Value Patterns	Template 1. CD Attribute Changes Template 13. OD Object States Changes	√ Attributes	√ Object States	√ Object States	√ Object States	√ Variables
Pattern 2. Operation Redundancy Check Pattern Pattern 4. Class with No Operation or Attribute Consistency Check Pattern Pattern 5. Class Element Redundancy Check Pattern Pattern 9. CD Operation Search Pattern Pattern 19. ADs Not Created Pattern Pattern 20. Activity Search Pattern Pattern 22. AD Elements Not Created Pattern Pattern 23. AD Action Search Pattern Pattern 33. SDs Not Created Pattern Pattern 34. SD Search Pattern Pattern 49. CD Create New Operation Patterns Pattern 59. CD Delete Operation Patterns Pattern 70. CD Modify Operation Property Patterns Pattern 71. CD Modify Operation Type Patterns Pattern 72. CD Modify Operation Visibility Patterns Pattern 73. Modify SD Name Patterns	Template 2. CD Operation Changes Template 13. OD Object States Changes	√ Operations	√ Object States	√ Activities and Sub Activities, Actions	√ Sequence diagrams states, Messages	√ Events

Patterns Provided	**Template**	**Affected Diagrams and Elements**				
		CD	**OD**	**AD**	**SD**	**SCD**
Pattern 74. Modify Operation Name Patterns Pattern 78. ADs Modify AD Name Pattern Pattern 79. SCDs Not Created Pattern Pattern 80. SCD Event Search Pattern Pattern 81. SCD Elements Not Created Pattern Pattern 82. SCD Action Search Pattern	**Previous page**					
Pattern 3. Class Redundancy Check Pattern Pattern 7. Check Object Name Pattern Pattern 10. CD Class Search Pattern Pattern 21. Objects Not in ADs Pattern Pattern 31. AD Object Search Pattern Pattern 35. Objects Not in SDs Pattern Pattern 44. SD Object Search Pattern Pattern 55. CD Delete Class Patterns Pattern 60. CD Modify Class Name Patterns Pattern 76. OD Delete Object Pattern Pattern 77. OD Modify Object Name Pattern	Template 3. CD Class Changes	√ Classes	√ Objects instances	√ Objects Instances,	√ Objects instances	√
Pattern 6. Class with No Relation Consistency Check Pattern Pattern 14. CD Generalization Search Pattern Pattern 52. CD Create Generalize Patterns Pattern 58. CD Delete Generalize Patterns Pattern 69. CD Modify Generalize Patterns	Template 4. CD Generalization/Class Inheritance Changes	√ Classes Relation/ inheritance Relations	√	√	√	√
Pattern 6. Class with No Relation Consistency Check Pattern Pattern 11. CD Association Search Pattern Pattern 12. CD Composition Search Pattern Pattern 13. CD Aggregation Search Pattern Pattern 14. CD Generalization Search Pattern	Template 5. CD Association Changes	√ Associations Aggregation Composition	√	√ Seq. of Activities, cntrl node, call behaviour	√ operators	√

Patterns Provided	**Template**	**Affected Diagrams and Elements**				
		CD	**OD**	**AD**	**SD**	**SCD**
Pattern 51. CD Create Association or Composition or Aggregation Patterns Pattern 53. CD Delete Aggregation Patterns Pattern 56. CD Delete Composition Patterns	**Previous page**					
Pattern 54. CD Delete Association Patterns	Template 6. CD Navigability Arrow Changes Template 27. AD Control Flow Changes Template 28. AD Object Flow Changes	√ Associations	√ Object Flow	√ Object and Control Flow	√ Object Flow	√
Pattern 61. CD Modify Association Destination Multiplicity Patterns Pattern 62. CD Modify Association Source Multiplicity Patterns	Template 8. CD Multiplicity Changes	√				
Pattern 63. CD Modify Role Name Patterns	Template 9. CD Role Name Changes	√				
Pattern 7. Check Object Name Pattern Pattern 15. Objects Not Created Pattern Pattern 16. Search Instance Name Pattern Pattern 17. Search Object Exists Pattern Pattern 18. Search Instance Class Pattern Pattern 21. Objects Not in ADs Pattern Pattern 35. Objects Not in SDs Pattern Pattern 44. SD Object Search Pattern Pattern 75. OD Create Object Pattern	Template 12. OD Object (Class instance) Changes Template 37. SD and CommD Object Changes		√ Object Instances	√ Object Instances	√ Object Instances	√
Pattern 32. AD Sub-Activity Search Pattern	Template 25. AD Sub-Activity/SCD Activity Changes			√ Activity/ Sub-Activity	√ SD Ref Operator	√

Patterns Provided	**Template**	**Affected Diagrams and Elements**				
		CD	**OD**	**AD**	**SD**	**SCD**
Pattern 23. AD Action Search Pattern Pattern 28. AD Call Behavioural Action Search Pattern Pattern 82. SCD Action Search Pattern	Template 26. UCD, SCD, and AD Action Changes			√ Action	√ Operators	√ Action
Pattern 27. AD Loop Search Pattern Pattern 39. SD Loop Search Pattern Pattern 84. SCD Loop Search Pattern	Template 31. AD, SD, and CommD Iteration /Loop Changes			√ Loop/ branches	√ Loop	√ Loop
Pattern 24. AD Fork Search Pattern Pattern 26. AD Join Search Pattern Pattern 29. AD Merge Search Pattern Pattern 30. AD Decision Search Pattern	Template 29. AD Control Nodes (Fork, Join, Merge, and Decision) Changes			√ Control nodes	√ operators	√
Pattern 25. AD Guard Search Pattern Pattern 41. SD Guard Search Pattern Pattern 83. SCD Guard Search Pattern	Template 35. SCD, AD, and SD Guard Condition Changes			√ Guard condition	√ Guard condition	√ Guard condition
Pattern 40. SD Massage Search Pattern	Template 38. SD Message Changes Template 39. SD Synchronous and Asynchronous Messages Changes	√ Comm method dynamic binding			√ synchronous asynchronous messages	√
Pattern 37. SD Alt Search Pattern Pattern 38. SD Par Search Pattern Pattern 42. SD Opt Search Pattern Pattern 43. SD Ref Search Pattern	Template 40. SD Operators (alt/ opt / ref / par) Changes			√ Cntrl node branches	√ Operators	√
Pattern 45. Changes History Selection Patterns Pattern 46. Store in File Pattern Pattern 47. Update New Version Pattern		Change Versions and History				
Pattern 36. SD Elements Not Created Pattern		Search about sequence diagram or activity diagram elements not created				

The proposed pattern design supports the automatic checking of consistency during the diagrams design process not just the checking of the consistency of the diagrams when they are updated. This can be considered a major advantage over the state-of-the-art approaches presented in Table 2.3. It also helps in solving the inconsistency detection problem. The search patterns proposed in this research can be used to detect inconsistencies before applying any diagrams changes. For example, the pattern design includes the following rule: Each message in a sequence diagram needs to have a corresponding operation that needs to be owned by the message receiver's class'. As shown in Figure 7.15, when there is any contradiction with this rule the change is rejected. The same things are applied for all the consistency rules proposed in this research.

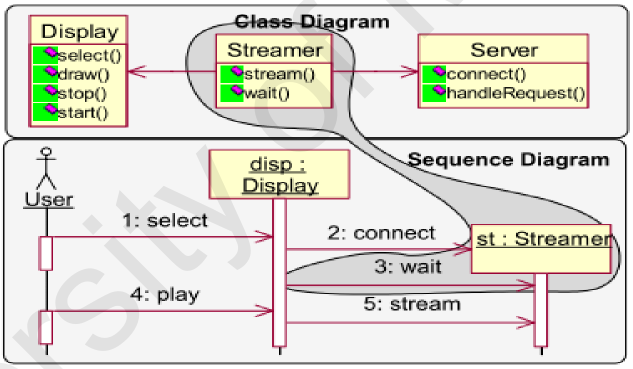

Figure 7.15: Example of Consistency between Diagrams

As illustrated above in (Table 7.9, Figure 7.10, and Figure 7.11), the metrics for quantifying the change impact/cost of the change in each coevolution pattern are based on the set of diagrams/diagrams elements affected by the change. The higher numbers explain the degree of coevolution between the diagrams also explain the high number of patterns proposed for the class diagram. The proposed coevolution patterns models were simulated in CPNs Tools as discussed in detail in Section 6.3. Additionally, these models can be exported to Java code.

7.3.1 Validation and Performance Analysis

The proposed framework validation and performance analysis is based on the CPNs Tools simulation and monitoring tool-boxes options, the results of which are shown in the following tables and figures. The monitoring and simulation tool-boxes allow checking at runtime that the system is behaving correctly.

A. Framework Validation

The simulation capabilities of CPNs Tools are used to execute the OOCPNs model over a set of test cases. The appropriate inputs for each test case were provided by placing tokens on the CPN places. The CPN model was then executed using the simulator toolbox to determine if the correct output was generated and if the correct logical paths were chosen. It should be noted that due to the state explosion problem it is very difficult to generate state space reports for the proposed framework. Therefore, in this research, the reachability of the places and transitions were detected through the use of marking size monitoring for all patterns as shown in Figure 7.17 to Figure 7.20.

B. Data Collector Monitoring:

Table 7.11 and Figure 7.16 illustrate the proposed framework model elements statistics. These statistics were derived from the CPNs Tools monitoring toolbox. These data also represent the model size or the scalability of the model.

Table 7.11: The Model Elements in the Proposed Framework Model

Diagram Element	Statistics (Number of Elements)
Places	2126
Place Instances	2274
Transitions	942
Transitions Instances	1418
Arcs	3638
Arcs Instances	4450
Pages	191

Diagram Element	Statistics (Number of Elements)
Pages Instances	267
Declaration (full CPN Tools declarations are provided in Appendix D)	262
Types	132
Variables	141

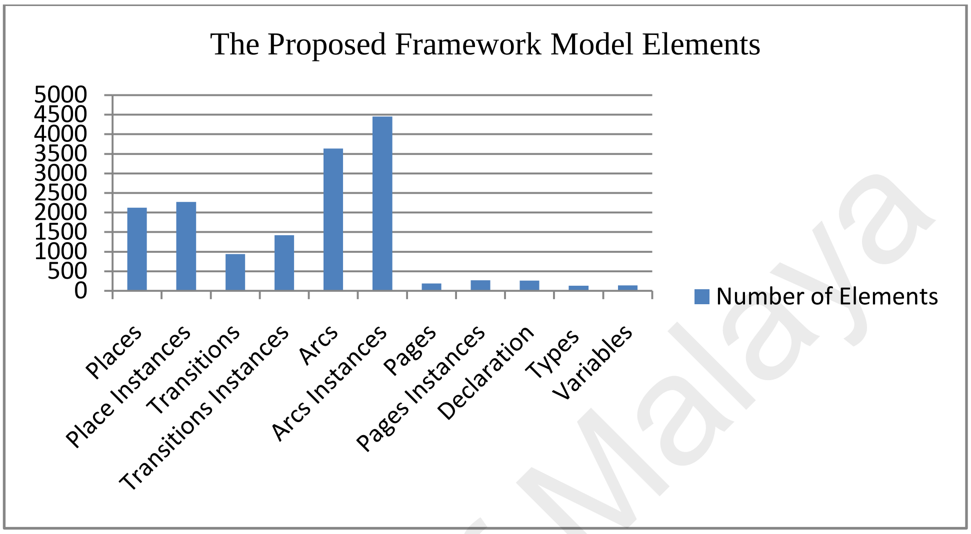

Figure 7.16: The Proposed Framework Model Elements-Model Size

C. Marking Size Monitoring:

Table 7.12 , Figure 7.17, and Figure 7.18 summarize the marking size monitoring data and data analysis results. The average metrics are calculated by Sum/Count.

Name	Count	Sum	Average
Class Diagrams	445	8	0.017937
Object Diagrams	246	19	0.076923
Activity Diagrams	503	11	0.021825
Sequence Diagrams	768	8	0.010296
Statechart Diagram	97	2	0.020619
Patterns	1301	1217	0.935434
Change History	1297	1206	0.929838

Table 7.12: Analysis of Marking Size Monitoring Data

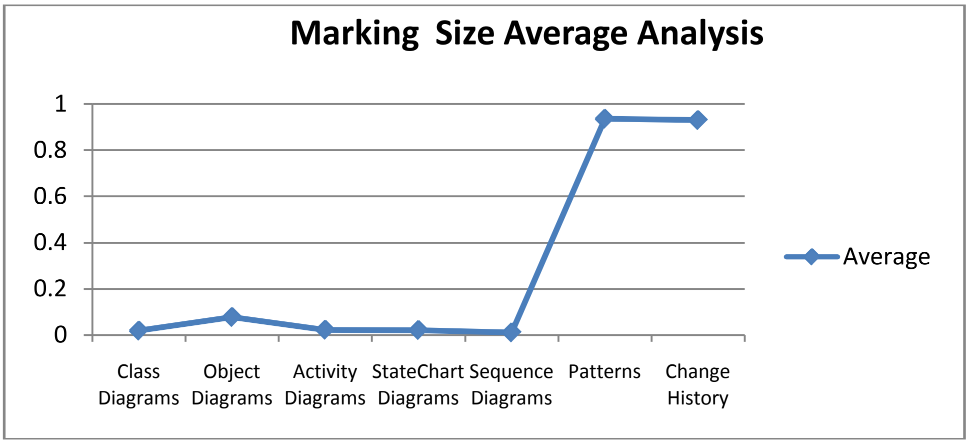

Figure 7.17: Analysis of Marking Size Monitoring Average

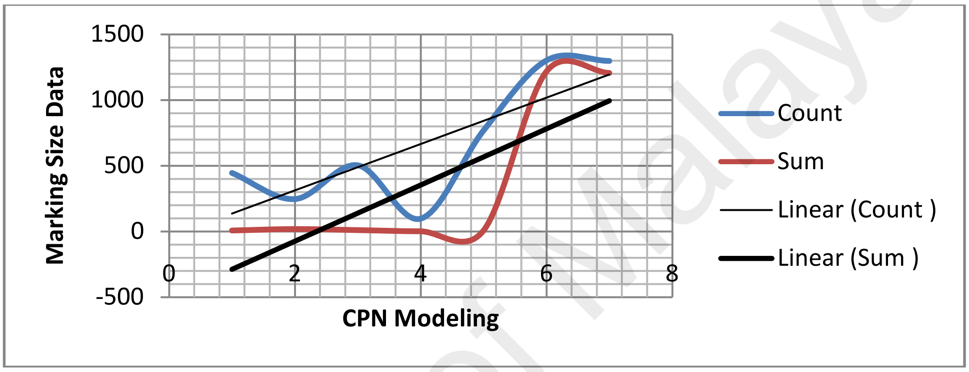

Figure 7.18: Analysis of Marking Size Monitoring

Detailed marking size monitoring analyses for each pattern are provided in Figure 7.19 and Figure 7.20.

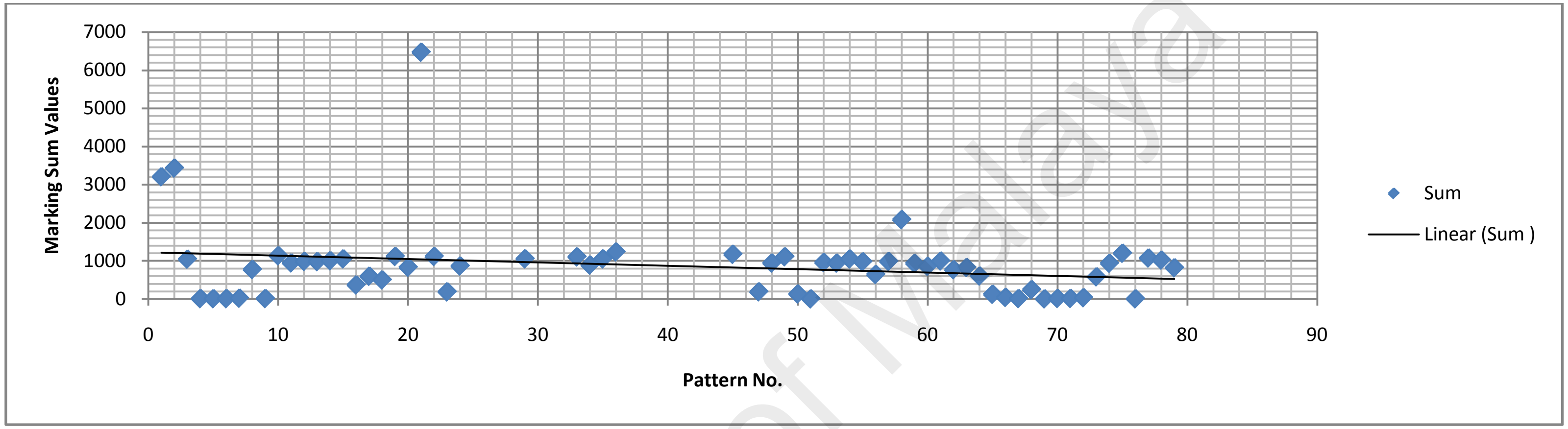

Figure 7.19: Analysis of Patterns Marking Size Sum

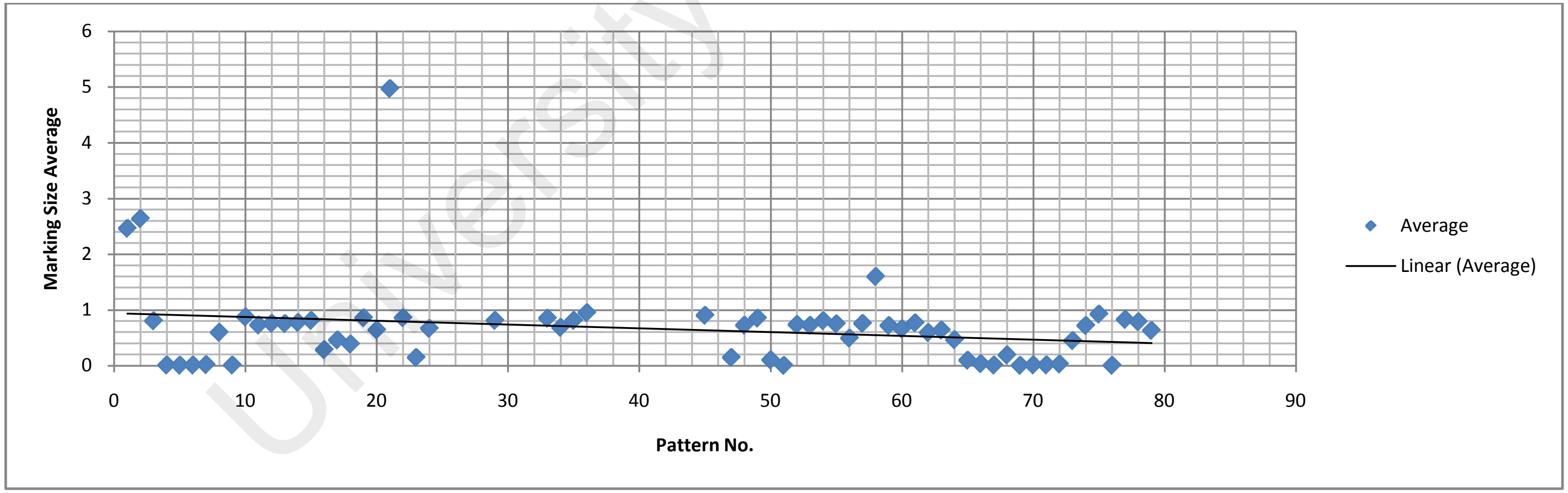

Figure 7.20: Analysis of Patterns Marking Size Average

7.3.2 Discussion

In related works (Kim, et al., 2007; NA Mulyar, 2009; Nataliya Mulyar & van der Aalst, 2005; N. C. Russell, 2007; Weber, et al., 2007; Wörzberger, et al., 2008) the patterns that are provided are specified only for modelling the business process and workflow software management system. On the other hand, the patterns approaches in (Gamma , et al., 1995; Gamma, et al., 2001) are used as design patterns. In contrast, the patterns in the framework proposed in this research can be used to deal with software changes in any OO diagrams design.

According to (Côté & Heisel, 2009), patterns exist not only as design patterns, but for every phase of software development, including requirements analysis, architectural design, implementation , and testing. The patterns in the proposed framework can also be applied to these phases in addition to the software maintenance phase. The proposed framework produces a precise set of dynamic impacts for UML diagrams by eliminating the changes through incremental consistency checks during the design stage and by identifying the change impact in the software maintenance/evolution stage.

in comparision with the state of the art approaches:

- Effectiveness and Soundness:
  - ✓ The proposed patterns help developers to build their models efficiently, while avoiding reinvention of already existing solutions of problems.
  - ✓ The proposed patterns express sound solutions for problems frequently recurring in a certain domain in a pattern format. Knowing a problem at hand, a developer can look up a solution for the problem in the pattern catalog, while spending less effort on the development and also ensuring the soundness of a solution.

  - ✓ This research classifies the patterns and analyses the relationships between the patterns to enable easy navigation through the patterns and this makes the evolution tasks easier.
  - ✓ The modularity in the hierarchical structure of the proposed framework reduces interdependencies between the model components, and facilitates easy maintenance and updates without impacting the entire model.
  - ✓ The change impact and traceability analysis rules are supported in the transformation of UML diagrams, this will improve the overall efficiency in software change management.
  - ✓ Not a comparison between two versions only.
- Maintainability:
  - ✓ Enhances the diagrams' change support through building a consistent OOCPNs model at the design time, and then applying the changes on the OOCPNs models. not just the checking of the consistency of the diagrams when they are updated.
  - ✓ This will provide incremental and automatic coevolution and consistency check.
  - ✓ Executable OOCPNs model - Incremental and Automatic correctness check using CPNs simulation and monitoring tools.
- Integrity:
  - ✓ Integrate the new changes with the current diagrams.
- Completeness and Functionality:
  - ✓ Cover all UML 2.3 diagrams in the proposed OOCPNs structure and in the proposed change impact and traceability analysis Templates.

7.4 Accomplishment of Research Objectives

The primary goal of this research was accomplished through the proposal of a new coevolution framework to enhance the representation capabilities of OO and CPNs modelling languages to support model changes. The proposed framework manages the coevolution between UML diagrams after each update operation, where UML diagrams are modelled from different perspectives using UML structural, behavioural, and interaction diagrams. The main objectives of this research were achieved and the research questions were answered as follows:

a. A new structure for the integration of UML and CPNs (Object Oriented Coloured Petri Nets (OOCPNs)) was proposed and evaluated. In this structure, transformation rules are applied between UML diagrams' elements and OOCPNs. The proposed structure also includes consistency and integrity rules that are applied when updating diagrams and diagram elements. This answers RQ1.
b. A set of change impact and traceability analysis templates for all types of change in most of the UML 2.3 diagrams was proposed and evaluated. The templates include, rules to maintain consistency and integrity. This answers part of RQ2.
c. A set of coevolution patterns to model and simulate the proposed diagrams changes was proposed and evaluated. The patterns include the change impact and traceability analysis templates for updating UML diagrams. This completely answers RQ2.
d. The development of the proposed coevolution framework answers RQ3.

To answer RQ4, the performance of the proposed coevolution framework was quantified through simulation statistics and a framework analysis which were provided in Sections 6.3 and 7.1 to 7.3.

7.5 Limitations of Research

The main limitations of this research are as follows:

1. The proposed framework is restricted on term of the range of UML diagrams supported in the patterns design (specifically class, object, activity, statechart, and sequence diagrams). Hence a more comprehensive framework is required to cover all diagrams.
2. This research does not cover all the possible inconsistency checking rules for all diagrams. This is because the research focuses on the most important diagrams elements and rules.
3. Although the proposed OOCPNs patterns describe the UML diagrams consistency problems and the solutions can be applied when modelling a wide range of systems, the applicability of these patterns is limited to the CPNs community because the implementation of the patterns is CPNs language dependent.

7.6 Chapter Summary

This chapter discussed the research findings in detail. This chapter presented the simulation methodology and some scenarios. Moreover, the framework results were analysed and discussed, including the proposed integration between UML and CPNs (i.e. the new OOCPNs structure including the transformation rules and the consistency rules), the proposed change impact template, and the proposed coevolution patterns. The next chapter will summarize the thesis outcomes and findings and will also highlight the research contributions and limitations. Finally, some conclusions are drawn and recommendations are made on some potential future research areas are highlighted.

CHAPTER 8: CONCLUSION AND FUTURE WORK

8.1 Thesis Summary

As software evolves, analysis and design models need be modified, accordingly. To cope with changes in the software process, in this research, a novel approach for a coevolution framework was proposed to manipulate the change effect in the UML diagrams' elements. In this framework, UML diagrams are modelled from different perspectives using UML structural, behavioural, and interaction diagrams. The proposed framework can be applied to detect the diagram elements affected by a change in a system design modelled using UML diagrams by utilising the proposed coevolution patterns. This framework can be used to control the evolution of UML diagrams by identifying and managing the model changes, ensuring the correctness and consistency of the models, identifying the impact of changes based on the relationships between diagrams, and analyzing the performance.

In addition, a set of model-based change impact and traceability analysis templates was proposed to determine and classify the types of changes in UML diagrams and their impact on other diagrams. The consistency between diagrams is checked according to the consistency and integrity rules provided in each template. This includes the vertical, horizontal, and evolutionary consistency types. Changes are modelled using coevolution patterns. CPNs Tools toolboxes are used to model and simulate the proposed framework.

This research also proposed a new structure for the mutual integration between UML diagrams and CPNs to support model changes. This structure combines the advantages of the formal and semi-formal modelling languages. The UML diagrams as a semi-formal modelling language are used to provide powerful structuring capabilities in the model design. The CPNs as a formal and executable modelling language describe the behaviour of the UML model formally. In addition, transformation rules are proposed to transform the UML diagrams into OOCPNs model. Moreover, rules to maintain the consistency and integrity of the OOCPNs model are proposed to support the model changes. The consistency and integrity rules are based on the UML diagrams relations and the proposed OOCPNs structure.

In this research, UML diagrams offered in UML 2.3 are supported in the transformation between UML diagrams into CPNs and in the proposed change impact and traceability analysis templates. The proposed coevolution patterns support the UML class, object, activity, statechart, and sequence diagrams because the coevolution between these diagrams is very high (the class diagram and object diagram represent the structured diagrams perspectives. The statechart diagram, activity diagram, and sequence diagram represents the behavioural and interaction diagrams perspectives). The proposed patterns support the checking of the consistency between UML diagrams during the design process not just checking of the consistency when the diagrams are updated. The coevolution is incremental; this means that if the Addition for a new diagram element is related to other diagrams elements it must exist, as shown in Figure 7.15 which provides an example of the incrementally consistency check. Incremental checking includes consistency and integrity rules.

8.2 Research Contributions and Significance

The new framework proposed in this thesis will be of assistance to software engineers because it is a systematic and methodical approach for change analysis and management.

This research started by addressing the transformation between UML diagrams and CPNs as well as consistency checking rules. Then, a set of change impact and traceability analysis templates for all types of change in UML diagrams was proposed, including rules to maintain consistency and integrity. Finally, a set of coevolution patterns was proposed to model and simulate the proposed framework, including the change impact and traceability analysis templates for updating OO diagrams. The proposed patterns were used to validate and verify the software model based on checking the correctness and complexity after updating the model using these patterns.

The proposed framework can be implemented for actual deployments in any system modelled using UML diagrams, such as those in large universities, industrial factories, large or small companies, to provide software model analysis and design. The proposed framework has the following benefits:

1. It enables comprehensive modelling for changes in UML diagrams;
2. It provides coevolution patterns and templates in OOCPNs for UML diagram changes. i.e. it improves pattern support in software analysis and design;
3. It provides a new structure for the integration between UML and CPNs to support model changes; and
4. It increases the structuring capabilities of CPNs.

8.3 Key Features and Outcomes

The main features and outcomes of this research are as follows:

Short-term outcomes:

- A coevolution framework to support UML diagram changes using OOCPNs patterns;
- A consistent integration of UML and CPNs based on the new proposed OOCPNs structure for the integration of UML and CPNs and the transformation rules applied between UML diagram elements and CPNs;
- A set of change impact and traceability analysis templates for all types of change in UML diagrams, including rules to maintain consistency and integrity;

- A set of coevolution patterns to model and simulate the proposed framework including the change impact and traceability analysis templates for updating UML diagrams; and
- Validation and verification of the software model based on checking the correctness and complexity after updating the model using coevolution patterns.

Long-term outcomes:

- Increased representation capability for UML modelling to support flexibility and adaptability in UML diagrams changes;
- An effective coevolution framework for dynamic changes in software models based on the integration of UML and CPNs modelling languages.

8.4 Recommendations for Future Research

The work done in this thesis could be extended in several directions:

- The proposed framework covers some of the UML diagrams in patterns design (namely class, object, activity, statechart, and sequence diagram). A more comprehensive framework could be attempted in a future research study.
- The provision of a software tool to automatically upload and transform UML diagrams to CPNs could also be developed.
- The limitations of this research mentioned in Section 7.5 could be addressed.
- Extending the research by cosidering the semantic meanings of the model.
- Considering the coevolution between models and the source code.
- Applying the proposed framework on realistec case studies.

REFERENCES

Abbasi, A. A. (2015). A pattern language for evolution reuse in component-based software architectures. Dublin City University.

Abma, B. (2009). Evaluation of requirements management tools with support for traceability-based change impact analysis. Master's thesis, University of Twente.

Aguilar-Saven, R. S. (2004). Business process modelling: Review and framework. International Journal of production economics, 90(2), 129-149.

Ajila, S. (1995). Software maintenance: an approach to impact analysis of objects change. Software: Practice and Experience, 25(10), 1155-1181.

Al-Khiaty, M. A.-R., & Ahmed, M. (2016). UML Class Diagrams: Similarity Aspects and Matching. Lecture Notes on Software Engineering, 4(1).

Alexander, C. (1979). The timeless way of building (Vol. 1): New York: Oxford University Press.

Ali, A., Boufares, F., & Abdellatif, A. (2006). Checking constraints consistency in UML class diagrams. Paper presented at the 2nd Information and Communication Technologies. ICTTA'06. .

Ali, H. O., Rozan, M. Z. A., & Sharif, A. M. (2012). Identifying challenges of change impact analysis for software projects. Paper presented at the International Conference on Innovation Management and Technology Research (ICIMTR).

Amar, B., Leblanc, H., Coulette, B., & Dhaussy, P. (2013). Automatic Co-evolution of Models Using Traceability Software and Data Technologies (pp. 125-139): Springer.

Ambler's, S. W. (2009). Introduction to the Diagrams of UML 2.0, Retrieved 2010, from http://www.agilemodeling.com/essays/umlDiagrams.htm

April, A., & Abran, A. (2012). Software maintenance management: evaluation and continuous improvement: Wiley-IEEE Computer Society Press.

Baader, F. (2003). The description logic handbook: theory, implementation, and applications: Cambridge university press.

Baresi, L. (2002). Some preliminary hints on formalizing UML with Object Petri Nets. Paper presented at the Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology.

Barr, P., & Pettis, J. (2007). UML 2.0 Diagrams, Petri Nets and Development of an Executable Architecture to predict performance. . Paper presented at the Conference on Systems Engineering Researh (CSER2007), Hoboken, NJ , USA.

Barros, J. P., & Gomes, L. (2004). On the use of coloured Petri nets for object-oriented design Applications and Theory of Petri Nets 2004 (pp. 117-136): Springer.

Barros, J. P., & Jorgensen, J. B. (2005). A CASE STUDY ON COLOURED PETRI NETS INl OBJECT-ORIENTED ANALYSIS AND DESIGN. Nordic Journal of Computing l2, 50, 22.

Bastide, R. (1995). Approaches in unifying Petri nets and the object-oriented approach. Paper presented at the 1st Workshop on Object-Oriented Programming and Models of Concurrency, within 16th international conference on applications and theory of Petri nets, ICATPN'95.

Bauskar, B. E., & Mikolajczak, B. (2006). Abstract node method for integration of object oriented design with colored Petri nets. Paper presented at the Third International Conference on Information Technology: New Generations. ITNG 2006.

Bennett, S., McRobb, S., & Farmer, R. (2010). Object-oriented systems analysis and design using UML: 4th Edition, McGraw-Hill Berkshire, UK.

Bhat, J. M., & Deshmukh, N. (2005). Methods for modeling flexibility in business processes. Paper presented at the Proceedings of the Sixth Workshop on Business Process Modeling, Development, and Support, BPMDS'05.

Biberstein, O., Buchs, D., & Guelfi, N. (1996). COOPN/2: A specification language for distributed systems engineering. Paper presented at the in DeVa 1st Year Report.

Bishop, L. (2004). Incremental impact analysis for object-oriented software. Lowa State University.

Bohner, S. A. (1996). Software change impact analysis: IEEE Computer Society Press
Bohner, S. A. (2002). Software change impacts-an evolving perspective. Paper presented at the Proceedings. International Conference on Software Maintenance.

Bokhari, A., & Poehlman, S. (2006). Translation of UML models to object coloured Petri nets with a view to analysis. Software Engineering and Knowledge Engineering (SEKE: 2006), 568-571.

Bolloju, N., Schneider, C., & Sugumaran, V. (2012). A knowledge-based system for improving the consistency between object models and use case narratives. Expert Systems with Applications, 39(10), 9398-9410.

Bouabana-Tebibel, T., & Belmesk, M. (2004). Formalization of UML object dynamics and behavior. Paper presented at the IEEE International Conference on Systems, Man and Cybernetics.

Bouabana-Tebibel, T., & Belmesk, M. (2005). Object-oriented workflow formalization. Paper presented at the Proc. of the 2nd South-East European Workshop on Formal Methods.

Bouabana-Tebibel, T., & Belmesk, M. (2007). An object-oriented approach to formally analyze the UML 2.0 activity partitions. Information and Software Technology, 49(9), 999-1016.

Bousse, E. (2012). Requirements management led by formal verification. Ms in cs, University of Rennes, 1.

Breivold, H. P., Crnkovic, I., & Larsson, M. (2012). A systematic review of software architecture evolution research. Information and Software Technology, 54(1), 16-40.

Briand, L. C., Labiche, Y., & O'sullivan, L. (2003). Impact analysis and change management of UML models. Paper presented at the International Conference on Software Maintenance. ICSM 2003. .

Briand, L. C., Labiche, Y., & Yue, T. (2009). Automated traceability analysis for UML model refinements. Information and Software Technology, 51(2), 512-527.

Bruckmann, T., & Gruhn, V. (2008a). Amabulo-a model architecture for business logic. Paper presented at the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. ECBS 2008. .

Brückmann, T., & Gruhn, V. (2008b). AMABULO Meta-Model: Formal Description and Formal Mapping into Coloured Petri Nets: University of Leipzig.

Bruegge, B. (2010). Object-Oriented Software Engineering: Using UML, Patterns and Java 3/E: Prentice Hall.

Calderon, M. E. (2005). Model transformation support for the analysis of large-scale systems. Master Thesis in Software Emgineering,
, Texas Tech University.

Calì, A., Calvanese, D., De Giacomo, G., & Lenzerini, M. (2002). A formal framework for reasoning on UML class diagrams Foundations of Intelligent Systems (pp. 503-513): Springer.

Campos, J., & Merseguer, J. (2006). On the integration of UML and Petri nets in software development Petri Nets and Other Models of Concurrency-ICATPN 2006 (pp. 19-36): Springer.

Capra, L., & Cazzola, W. (2007). A reflective PN-based approach to dynamic workflow change. Paper presented at the International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. SYNASC.

Chang, Y.-L., Chen, S., Chen, C.-C., & Chen, I. (2000). Workflow process definition and their applications in e-commerce. Paper presented at the International Symposium on Multimedia Software Engineering. .

Chen, C.-Y., & Chen, P.-C. (2009). A holistic approach to managing software change impact. Journal of Systems and Software, 82(12), 2051-2067.

Chen, C.-Y., She, C.-W., & Tang, J.-D. (2007). An object-based, attribute-oriented approach for software change impact analysis. Paper presented at the IEEE International Conference on Industrial Engineering and Engineering Management.

Chen, S. M. (2000). Using UML and Petri Nets for Workflow Process Definition. Master Thesis, National Central University- Computer Science and Information Engineering.

Chiorean, D., Paşca, M., Cârcu, A., Botiza, C., & Moldovan, S. (2004). Ensuring UML models consistency using the OCL Environment. Electronic Notes in Theoretical Computer Science, 102, 99-110.

Chiorean, D. I., Petrascu, V., & Petrascu, D. (2008). How my favorite tool supporting OCL must look like. Electronic Communications of the EASST, 15.

Chukwuogo, B. I. (2007). SCALBAILITY IN ANALYSIS OF SOFTWARE ARCHITECTURE. Master Thesis, Texas Tech University.

Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. Paper presented at the 12th International IEEE Enterprise Distributed Object Computing Conference. EDOC'08.

Costanza, P. (2001). Dynamic object replacement and implementation-only classes. Paper presented at the 6th International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP.

Côté, I., & Heisel, M. (2009). Supporting Evolution by Models, Components, and Patterns. Paper presented at the Proceedings of the 1. Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): "Design for Future - Langlebige Softwaresysteme".

Curtis, H. T., Clarence, S. L., & Ying, K. L. (2005). Object-Oriented Technology: From Diagram to Code with Visual Paradigm for UML: McGraw-Hill.

D'Hondt, T., De Volder, K., Mens, K., & Wuyts, R. (2002). Co-evolution of object-oriented software design and implementation Software Architectures and Component Technology (pp. 207-224): Springer.

Dadam, P., & Reichert, M. (2009). The ADEPT project: a decade of research and development for robust and flexible process support. Computer Science-Research and Development, 23(2), 81-97.

DAMIANO, T., LABICHE, Y., & GENERO, M. (2015). A systematic identification of consistency rules for UML diagrams. Carleton University, Technical Report SCE-15-01

Dang, D.-H., & Gogolla, M. (2016). An OCL-Based Framework for Model Transformations. VNU Journal of Science: Computer Science and Communication Engineering, 32(1).

De Lucia, A., Fasano, F., & Oliveto, R. (2008). Traceability management for impact analysis. Paper presented at the Frontiers of Software Maintenance, 2008. FoSM 2008.

Demuth, A., Riedl-Ehrenleitner, M., Lopez-Herrejon, R. E., & Egyed, A. (2016). Co-evolution of metamodels and models through consistent change propagation. Journal of Systems and Software, 111, 281-297.

Dubauskaite, R., & Vasilecas, O. (2013). Method on Specifying Consistency Rules among Different Aspect Models, expressed in UML. Electronics and Electrical Engineering, 19(3), 77-81.

Egyed, A. (2006). Instant consistency checking for the UML. Paper presented at the Proceedings of the 28th international conference on Software engineering.

Egyed, A. (2007a). Fixing inconsistencies in UML design models. Paper presented at the 29th International Conference on Software Engineering. ICSE 2007. .

Egyed, A. (2007b). Uml/analyzer: A tool for the instant consistency checking of uml models. Paper presented at the 29th International Conference on Software Engineering. ICSE 2007. .

Egyed, A. (2011). Automatically detecting and tracking inconsistencies in software design models. Software Engineering, IEEE Transactions on, 37(2), 188-204.

Einarsson, H. ó., & Neukirchen, H. (2012). An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations. Paper presented at the Proceedings of the Fifth Workshop on Refactoring Tools.

Ekanayake, E., & Kodituwakku, S. R. (2015). Consistency checking of UML class and sequence diagrams. Paper presented at the 8th International Conference on Ubi-Media Computing (UMEDIA). .

Elaasar, M., & Briand, L. (2004). An overview of UML consistency management. Carleton University, Canada, Technical Report SCE-04-18.

Elkoutbi, M., & Keller, R. K. (2000). User interface prototyping based on UML scenarios and high-level Petri nets Application and Theory of Petri Nets 2000 (pp. 166-186): Springer.

Emadi, S., & Shams, F. (2008). From UML component diagram to an executable model based on Petri nets. Paper presented at the International Symposium on Information Technology. ITSim 2008. .

Emadi, S., & Shams, F. (2009). Transformation of usecase and sequence diagrams to petri nets. Paper presented at the ISECS International Colloquium on Computing, Communication, Control, and Management. CCCM 2009. .

Engels, G., Küster, J. M., Heckel, R., & Groenewegen, L. (2001). A methodology for specifying and analyzing consistency of object-oriented behavioral models. Paper presented at the ACM SIGSOFT Software Engineering Notes.

Esser, R. (1997). An object oriented Petri net language for embedded system design. Paper presented at the Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [incorporating Computer Aided Software Engineering].

Etien, A., Rolland, C., & Salinesi, C. (2004). Overview of a Gap-driven Evolution Process. Paper presented at the Proceedings of Australian Workshop on Requirements Engineering, AWRE.

Etien, A., & Salinesi, C. (2005). Managing requirements in a co-evolution context. Paper presented at the 13th IEEE International Conference on Requirements Engineering. .

Fernandes, J. M., Tjell, S., Jorgensen, J. B., & Ribeiro, Ó. (2007). Designing tool support for translating use cases and UML 2.0 sequence diagrams into a coloured Petri net. Paper presented at the Sixth International Workshop on Scenarios and State Machines. SCESM'07: ICSE Workshops 2007. .

Fowler, M. (1999). Refactoring: improving the design of existing code: Addison-Wesley Professional.

Fowler, M. (2004). UML Distilled: A Brief Guide to the Standard Object Modeling Languange: Addison-Wesley Professional.

Fryz, L., & Kotulski, L. (2007). Assurance of system consistency during independent creation of UML diagrams. Paper presented at the 2nd International Conference on Dependability of Computer Systems. DepCoS-RELCOMEX'07. .

Gamma , E., Helm, R., Johnson , R., & Vlissides, J. (1995). Design Patterns: Rlements of Reusable Object Oriented Software: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2001). Design patterns: Abstraction and reuse of object-oriented design: Springer.

García, J., Diaz, O., & Azanza, M. (2013). Model transformation co-evolution: A semi-automatic approach Software Language Engineering (pp. 144-163): Springer.

Garrido, J. L., & Gea, M. (2002). A Coloured Petri Net Formalisation for a UML-Based Notation Applied to Cooperative System Modelling Interactive Systems: Design, Specification, and Verification (pp. 16-28): Springer.

Gethers, M., Dit, B., Kagdi, H., & Poshyvanyk, D. (2012). Integrated impact analysis for managing software changes. Paper presented at the 34th International Conference on Software Engineering (ICSE).

Ghosh, S., Sharma, H., & Mohabay, V. (2011a). Software change management–Technological dimension. International Journal of International Journal of o Smart Home Smart Home Smart Home 5(2).

Ghosh, S., Sharma, H., & Mohabay, V. (2011b). Analysis and Modelling of Change Management Process Model. International Journal of Software Engineering and Its Applications, 5(2).

Ghosh, S., Sharma, H., & Mohabay, V. (2011c). A Study of Software Change Management Problem. International Journal of Database Theory and Application 4(3).

Gómez-Martínez, E., & Merseguer, J. (2006). ArgoSPE: Model-based software performance engineering Petri Nets and Other Models of Concurrency-ICATPN 2006 (pp. 401-410): Springer.

Gongzheng, L., & Guangquan, Z. (2010). An approach to check the consistency between the UML 2.0 dynamic diagrams. Paper presented at the 5th International Conference on Computer Science and Education (ICCSE).

Grossmann, G., Mafazi, S., Mayer, W., Schrefl, M., & Stumptner, M. (2015). Change propagation and conflict resolution for the co-evolution of business processes. International Journal of Cooperative Information Systems, 24(01), 1540002.

Guerra, E., & de Lara, J. (2003). A framework for the verification of UML models. Examples using petri nets. Paper presented at the Proc. JISBD.

Halle, B. v., & Ronald, G. (2001). Business rules applied: building better systems using the business rules approach: John Wiley & Sons, Inc.

Hammad, M., Collard, M. L., & Maletic, J. I. (2010). Measuring class importance in the context of design evolution. Paper presented at the IEEE 18th International Conference on Program Comprehension (ICPC).

Hanish, A. A., & Dillon, T. S. (1997). Object-oriented behaviour modelling for real-time design. Paper presented at the Third International Workshop on Object-Oriented Real-Time Dependable Systems.

He, X. (2000). Formalizing UML class diagrams-a hierarchical predicate transition net approach. Paper presented at the The 24th Annual International Computer Software and Applications Conference. COMPSAC 2000. .

Herzig, S., Qamar, A., Reichwein, A., & Paredis, C. J. (2011). A conceptual framework for consistency management in model-based systems engineering. Paper presented at the Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2011.

Holliday, M. A., & Vernon, M. K. (1987). A generalized timed Petri net model for performance analysis. Software Engineering, IEEE Transactions on(12), 1297-1310.

Hollingsworth, D., & Hampshire, U. (1993). Workflow management coalition the workflow reference model Workflow Management Coalition (pp. 68).

Hong, J.-E., & Bae, D.-H. (2001). High-level Petri net for incremental analysis of object-oriented system requirements. IEE Proceedings-Software, 148(1), 11-18.

Hongmei, G., Biqing, H., & Shouju, R. (2000). A UML and Petri Nets Integrated Modeling Method for Business Processes in Virtual Enterprises. Paper presented at the Bringing Knowledge to Business Processes: Papers from the 2000 AAAI Symposium, March 20-22, Stanford, California.

Hößler, J., Soden, M., & Eichler, H. (2005). Coevolution of models, metamodels and transformations. Models and Human Reasoning. Wissenschaft und Technik Verlag, Berlin, 129-154.

Hu, Z., & Shatz, S. M. (2004). Mapping UML diagrams to a Petri net notation for system simulation. Paper presented at the 16th Int. Conf. on Software Engineering & Knowledge Engineering (SEKE 2004).

Huang, L., & Song, Y.-T. (2007). Precise dynamic impact analysis with dependency analysis for object-oriented programs. Paper presented at the 5th ACIS International Conference on Software Engineering Research, Management & Applications. SERA 2007.

Huzar, Z., Kuzniarz, L., Reggio, G., & Sourrouille, J. L. (2005). Consistency problems in UML-based software development UML Modeling Languages and Applications (pp. 1-12): Springer.

IBMSoftware. (2011). IBM Software. Retrieved from http://www-01.ibm.com/software/awdtools/developer/rose/

Ibrahim, N., Ibrahim, R., Saringat, M. Z., Mansor, R. D., & Herawan, T. (2013). Use case driven based rules in ensuring consistency of UML model. AWERProcedia Information Technology and Computer Science, 1.

Ibrahim, S., Idris, N. B., Munro, M., & Deraman, A. (2005). A requirements traceability to support change impact analysis. Asian Journal of Information Tech, 4(4), 345-355.

Isaac, S., & Navon, R. (2013). A graph-based model for the identification of the impact of design changes. Automation in Construction, 31, 31-40.

Ivkovic, I., & Kontogiannis, K. (2004). Tracing evolution changes of software artifacts through model synchronization. Paper presented at the 20th IEEE International Conference on Software Maintenance. Proceedings. .

Jaafar, F. (2012). On the analysis of evolution of software artefacts and programs. Paper presented at the Proceedings of the 2012 International Conference on Software Engineering.

Jennings, N. R., Faratin, P., Norman, T. J., O'Brien, P., Odgers, B., & Alty, J. L. (2000). Implementing a business process management system using ADEPT: A real-world case study. Applied Artificial Intelligence, 14(5), 421-463.

Jensen, K. (1992). Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. . Monographs in Theoretical Computer Science, Springer-Verlag, 1.

Jensen, K. (1994). An introduction to the theoretical aspects of coloured petri nets: Springer.

Jensen, K. (1998). An introduction to the practical use of coloured petri nets Lectures on Petri Nets II: Applications (pp. 237-292): Springer.

Jensen, K., & Kristensen, L. M. (2009). Coloured Petri nets: modelling and validation of concurrent systems: Springer Publishing Company, Incorporated.

Jensen, K., Kristensen, L. M., & Wells, L. (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer, 9(3-4), 213-254.

Jönsson, P. (2005). Impact Analysis: Organisational Views and Support Techniques.

Jørgensen, J. B. (2003). Coloured Petri nets in development of a pervasive health care system Applications and Theory of Petri Nets 2003 (pp. 256-275): Springer.

Kagdi, H., Gethers, M., & Poshyvanyk, D. (2012). Integrating conceptual and logical couplings for change impact analysis in software. Empirical Software Engineering, 1-37.

Kchaou, D., Bouassida, N., & Ben-Abdallah, H. (2016). Managing the Impact of UML Design Changes on Their Consistency and Quality. Arabian Journal for Science and Engineering, 1-19.

Keller, R. K., Shen, X., & Bochmann, G. v. (1994). Macronet-A Simple, yet Expressive and Flexible Formalism for Business Modelling. Paper presented at the Proceedings of the Workshop on Computer-Supported Cooperative Work, Petri Nets and Related Formalisms during the 15th International Conference on Application and Theory of Petri Nets. Zaragoza, Spain.

Khadka, B. (2007). Transformation of Live Sequence Charts to Colored Petri Nets. A Masters Project Report, University Of Massachusetts Dartmouth.

Khalil, A., & Dingel, J. (2013). Supporting the Evolution of UML Models in Model Driven Software Development: A Survey: Technical Report 2013-602 , School of Computing, Queen’s University

Khan, A. H., & Porres, I. (2015). Consistency of UML class, object and statechart diagrams using ontology reasoners. Journal of Visual Languages & Computing, 26, 42-65.

Kim, D., Kim, M., & Kim, H. (2007). Dynamic business process management based on process change patterns. Paper presented at the International Conference on Convergence Information Technology. .

Knolmayer, G., Endl, R., & Pfahrer, M. (2000). Modeling processes and workflows by business rules Business Process Management (pp. 16-29): Springer.

Koci, R., Janousek, V., & Zboril, F. (2008). Object Oriented Petri Nets Modelling Techniques Case Study. Paper presented at the Second UKSIM European Symposium on Computer Modeling and Simulation. EMS'08. .

Koomsub, D. (1999). A Case Study of Change Management of ERP Implementation Project Using SAP R/3. Thailand, Independent Study Project.

Kordic, V. (2008). Petri Net, Theory and Applications: InTech.

Kowalkiewicz, M., Lu, R., Bäuerle, S., Krümpelmann, M., & Lippe, S. (2008). Weak dependencies in business process models. Business Information Systems, 177-188.

Kradolfer, M. (2000). A workflow metamodel supporting dynamic, reuse-based model evolution. PhD thesis, Department of Information Technology, University of Zurich, Switzerland.

Krena, B., & Vojnar, T. (2001). Type Analysis in Object-Oriented Petri Nets. Proceedings of ISM’01 Hradec nad moranici Czech Republic, 173-180.

Kung, D., Gao, J., Hsia, P., Wen, F., Toyoshima, Y., & Chen, C. (1994). Change impact identification in object oriented software maintenance. Paper presented at the International Conference on Software Maintenance. .

Kurt, J. (1997). Coloured Petri nets: Basic concepts, analysis methods and practical use. EATCS Monographs on Theoretical Computer Science. 2nd edition, Berlin: Springer-Verlag.

Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schoenboeck, J., Schwinger, W., & Wimmer, M. (2015). Systematic Co-evolution of OCL expressions. 11th APCCM, 27, 30.

Lakos, C. (2001). Object oriented modelling with object Petri nets Concurrent object-oriented programming and petri nets (pp. 1-37): Springer.

Lakos, C., & Keen, C. (1994). LOOPN++: A new language for object-oriented Petri nets: Department of Computer Science, University of Tasmania.

Lakos, C., Keen, C., & Hobart, T. (1991). Simulation with object-oriented petri nets. Paper presented at the Australian Software Engineering Conference, Sydney.

Lam, W., Shankararaman, V., Jones, S., Hewitt, J., & Britton, C. (1998). Change analysis and management: a process model and its application within a commercial setting. Paper presented at the IEEE Workshop on Application-Specific Software Engineering Technology. ASSET-98. Proceedings. .

Langer, P., Mayerhofer, T., Wimmer, M., & Kappel, G. (2014). On the Usage of UML: Initial Results of Analyzing Open UML Models. Modellierung 19, 21.

Langhammer, M. (2013). Co-evolution of component-based architecture-model and object-oriented source code. Paper presented at the Proceedings of the 18th international doctoral symposium on Components and architecture.

Lassen, K. B. (2007). Translating UML 2.0 sequence charts into coloured Petri net using process mining: Technical report, Department of Computer Science at the University of Aarhus.

Le Bail, J., Alla, H., & David, R. (1991). Hybrid petri nets. Paper presented at the European Control Conference.

Lee, M. L. (1998). Change impact analysis of object-oriented software. George Mason University.

Lehnert, S. (2011). A review of software change impact analysis. Ilmenau University of Technology, Tech. Rep.

Lehnert, S., & Riebisch, M. (2013). Rule-Based Impact Analysis for Heterogeneous Software Artifacts. Paper presented at the 17th European Conference on Software Maintenance and Reengineering (CSMR).

Lewis, G. A. (1996). Producing network applications using object-oriented petri nets. Master Thesis, University of Tasmania.

Li, B., Sun, X., Leung, H., & Zhang, S. (2012). A survey of code -based cha analysis techniques. Software Testing, Verification and Reliability.

Lian-Zhang, Z., & Fan-Sheng, K. (2012). Automatic Conversion from UML to CPN for Software Performance Evaluation. Procedia Engineering, 29, 2682-2686.

Liles, S. W. (2008). On the characterization and analysis of system of systems architectures. PhD Thesis, George Mason University.

Liu, X. (2013). Identification and Check of Inconsistencies between UML Diagrams. Journal of Software Engineering and Applications, 6, 73-77.

Liui, X., Yin, G., & Zhang, Z. (2008). A Kind of Object-Oriented Petri Net and Its Application. Paper presented at the International Conference on Internet Computing in Science and Engineering. ICICSE'08. .

Lu, R. (2008). Constraint-Based Flexible Business Process Management. PhD Thesis, School of Information Technology and Electrical Engineering, University of Queensland.

Lu, R., & Sadiq, S. (2007). A survey of comparative business process modeling approaches. Business Information Systems, 82-94.

Lucas, F. J., Molina, F., & Toval, A. (2009). A systematic review of UML model consistency management. Information and Software Technology, 51(12), 1631-1645.

Mäder, P., Gotel, O., & Philippow, I. (2009). Enabling automated traceability maintenance through the upkeep of traceability relations. Paper presented at the Model Driven Architecture-Foundations and Applications.

MagicDraw. (2009). MagicDraw-UML. Retrieved from www.magicdraw.com/

Mahmood, Z., & Mahmood, R. B. T. (2015). Category, Strategy and Validation of Software Change Impact Analysis. International Journal Of Engineering And Computer Science, 11(4), 11126-11128

Malabarba, S., Pandey, R., Gragg, J., Barr, E., & Barnes, J. F. (2000). Runtime support for type-safe dynamic Java classes: Springer.

Maqbool, S. (2005). Transformation of a core scenario model and activity diagrams into petri nets. Master Thesis, University of Ottawa.

Marzeta, R. (2007). Specification of design and verification of service-oriented systems. Master's thesis, Technical University of Denmark, Informatics and Mathematical Modelling, DTU, IMM Publication.

Meijers, M. (1996). Tool support for object-oriented design patterns. Dept. of Computer Science INF-SCR-96-28, Utrecht University (August 1996).

Mendling, J., La Rosa, M., & ter Hofstede, A. H. (2008). Correctness of business process models with roles and objects: QUT ePrints Technical Report #13172, Queensland University of Technology,Australia

Mens, T., Van Der Straeten, R., & Simmonds, J. (2005a). A framework for managing consistency of evolving UML models. Software Evolution with UML and XML, 1-31.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., & Jazayeri, M. (2005b). Challenges in software evolution. Paper presented at the Eighth International Workshop on Principles of Software Evolution.

Merseguer, J. e., & Campos, J. (2003). Software Performance Modeling using UML and Petri nets, MASCOTS Tutorials: 265-289.

Mikolajczak, B., & Sefranek, C. (2003). Integrating object-oriented design with Petri nets-case study of ATM system. Paper presented at the IEEE International Conference on Systems, Man and Cybernetics. .

Milanovic, M., Gasevic, D., & Wagner, G. (2008). Combining rules and activities for modeling service-based business processes. Paper presented at the 12th Enterprise Distributed Object Computing Conference Workshops.

Millan, T., Sabatier, L., Le Thi, T.-T., Bazex, P., & Percebois, C. (2009). An OCL extension for checking and transforming UML models. Paper presented at the International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'09), Cambridge, United Kingdom.

Miller, R. (2003). Practical UML: A hands-on introduction for developers. White Paper, Borland Developer Network.

Miyamoto, T., & Kumagai, S. (2005). A survey of object-oriented Petri nets and analysis methods. IEICE transactions on fundamentals of electronics, communications and computer sciences, 88(11), 2964-2971.

Miyamoto, T., & Kumagai, S. (2007). Application of Object-Oriented Petri Nets to Industrial Electronics. Paper presented at the 33rd Annual Conference of the IEEE Industrial Electronics Society. IECON 2007. .

Mohan, K., Xu, P., Cao, L., & Ramesh, B. (2008). Improving change management in software development: Integrating traceability and software configuration management. Decision Support Systems, 45(4), 922-936.

Motameni, H., Movaghar, A., Shirazi, B., Aminzadeh, M., & Samadi, H. (2008). Analysis Software with an object-oriented Petri net model. World Applied Sciences Journal, 3(4), 565-576.

Mulyar, N. (2009). Patterns for process-aware information systems: an approach based on colored Petri nets. PhD Thesis, Eindhoven: Technische Universiteit Eindhoven.

Mulyar, N., & van der Aalst, W. M. (2005). Towards a pattern language for colored petri nets. Paper presented at the sixth workshop and tutorial on practical use of coloured Petri nets and the CPN tools.

Murata, T. (1989). Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4), 541-580.

Murphy, G. C., Notkin, D., & Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. Paper presented at the ACM SIGSOFT Software Engineering Notes.

Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. Software Engineering, IEEE Transactions on, 27(4), 364-380.

Niepostyn, S. J. (2015). The Sufficient Criteria For Consistent Modelling Of The Use Case Realization Diagrams With A New Functional-Structure-Behaviour UML Diagram. Przegląd Elektrotechniczny Sigma NOT(2), 31-35.

Niu, J., Zou, J., & Ren, A. (2003). OOPN: Object-oriented Petri Nets and Its Integrated Development Environment. Paper presented at the Proceedings of the Software Engineering and Applications, SEA.

Nurcan, S. (2008). A survey on the flexibility requirements related to business processes and modeling artifacts. Paper presented at the Proceedings of the 41st Annual Hawaii International Conference on System Sciences.

OMG. (2004). Business Process Definition Metamodel. Version 1.0.2. Revised submission bei/2004-01-02.

OMG. (2010). Documents associated with UML Version 2.3, Retrieved from http://www.omg.org/spec/UML/2.3/.

Ossami, D. D. O., Jacquot, J.-P., & Souquières, J. (2005). Consistency in UML and B multi-view specifications. Integrated Formal Methods 386-405.

Ouardani, A., Esteban, P., Paludetto, M., & Pascal, J.-C. (2006). A Meta-modeling Approach for Sequence Diagrams to Petri Nets Transformation within the requirements validation process. Paper presented at the Proceedings of the European Simulation and Modeling Conference.

Owen, M., & Raj, J. (2003). BPMN and business process management. Introduction to the New Business Process Modeling Standard.

Park, S., Kim, H., & Bae, D.-H. (2009). Change impact analysis of a software process using process slicing. Paper presented at the 9th International Conference on Quality Software. QSIC'09. .

Pesic, M., & van der Aalst, W. M. (2006). A declarative approach for flexible business processes management. Paper presented at the Business Process Management Workshops.

Pnueli, A. (1977). The temporal logic of programs. Paper presented at the 18th Annual Symposium on Foundations of Computer Science

Podgurski, A., & Clarke, L. (1990). A formal model of program dependencies and its implication for software testing, debugging and maintenance. IEEE Transactions on Software Engineering, 16(9), 352-357.

Protic, Z. (2011). Configuration management for models: Generic methods for model comparison and model co-evolution. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.

Puczynski, P. J. (2012). Checking consistency between interaction diagrams and state machines in UML models. Technical University of Denmark.

Puissant, J. P. (2012). Resolving Inconsistencies in Model-Driven Engineering using Automated Planning. PhD thesis, Universit de Mons.

Puissant, J. P., Van Der Straeten, R., & Mens, T. (2013). Resolving model inconsistencies using automated regression planning. Software & Systems Modeling, 1-21.

Rajabi, B. A., & Lee, S. P. (2009a). Change management in business process modeling survey. Paper presented at the ICIME'09. International Conference on Information Management and Engineering. .

Rajabi, B. A., & Lee, S. P. (2009b). A Study of the Software Tools Capabilities in Translating UML Models to PN Models. International Journal of Intelligent Information Technology Application (IJIITA), , 2(5), 224-228.

Rajabi, B. A., & Lee, S. P. (2010). Modeling and analysis of change management in dynamic business process. International Journal of Computer and Electrical Engineering, 2(1), 181-189.

Rajabi, B. A., & Lee, S. P. (2014). Consistent Integration between Object Oriented and Coloured Petri Nets Models. The International Arab Journal of Information Technology, 11(4).

Rasch, H., & Wehrheim, H. (2003). Checking consistency in UML diagrams: Classes and state machines Formal Methods for Open Object-Based Distributed Systems (pp. 229-243): Springer.

Redding, G. M. (2009). Object-centric process models and the design of flexible processes. PhD Thesis, Faculty of Science and Technology, Queensland University of Technology, Brisbane, Australia.

Reder, A., & Egyed, A. (2012). Incremental consistency checking for complex design rules and larger model changes Model Driven Engineering Languages and Systems (pp. 202-218): Springer.

Reder, A., & Egyed, A. (2013). Determining the Cause of a Design Model Inconsistency. IEEE Transactions on Software Engineering, 1.

Reggio, G., Leotta, M., Ricca, F., & Clerissi, D. (2013). What are the used UML diagrams? A Preliminary Survey. Paper presented at the EESSMOD@ MoDELS.

Reichert, M., & Dadam, P. (1998). ADEPTflex—Supporting dynamic changes of workflows without losing control. Journal of Intelligent Information Systems, 10(2), 93-129.

Reichert, M., & Dadam, P. (2009). Enabling Adaptive Process-aware Information Systems with ADEPT2, In Handbook of Research on Business Process Modeling: Information Science Reference, Hershey, New York.

Reichert, M., Rinderle, S., Kreher, U., & Dadam, P. (2005). Adaptive Process Management with ADEPT2 Paper presented at the Proceedings of the 21st International Conference on Data Engineering

Ribeiro, O. R., & Fernandes, J. M. (2006). Some rules to transform sequence diagrams into coloured Petri nets. Paper presented at the 7th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (CPN 2006).

Russell, N., van der Aalst, W. M., Ter Hofstede, A. H., & Wohed, P. (2006). On the suitability of UML 2.0 activity diagrams for business process modelling. Paper presented at the Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling-Volume 53.

Russell, N. C. (2007). Foundations of process-aware information systems. PhD, Queensland University of Technology, Brisbane, Australia.

Saif, N., Razzaq, N., Rehman, S. U., Javed, A., & Ahmad, B. (2013). The Concept of Change Management in Today’s Business World. Information and Knowledge Management, 3, 28-33.

Saldhana, J., & Shatz, S. M. (2000). Uml diagrams to object petri net models: An approach for modeling and analysis. Paper presented at the International Conference on Software Engineering and Knowledge Engineering.

Salinesi, C., Etien, A., & Wäyrynen, J. (2004). Towards a Systematic Propagation of Evolution Requirements in IS Adaptation Projects. Paper presented at the Proceeding of Australian Conference on Information System ACIS.

Sapna, P., & Mohanty, H. (2007). Ensuring consistency in relational repository of UML models. Paper presented at the 10th International Conference on Information Technology,(ICIT 2007).

Scheer, A. W. (1994). ARIS toolset: a software product is born. Information Systems, 19(8), 607-624.

Scheer, A. W. (2000). ARIS: business process modeling: Springer.

Sharaff, A. (2013). A Methodology for Validation of OCL Constraints Using Coloured Petri Nets. International Journal of Scientific & Engineering Research, 4(1).

Shengyuan, W., & Yuan, D. (2007). Improving Combinability of Petri Nets with Inheritance, Aggregation and Association. Paper presented at the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. TASE'07. .

Shin, M. E., Levis, A. H., & Wagenhals, L. W. (2003). Transformation of UML-based system model to design/CPN model for validating system behavior. Paper presented at the Proc. of the 6th Int. Conf. on the UML/Workshop on Compositional Verification of the UML Models.

Shin, M. E., Levis, A. H., Wagenhals, L. W., & Kim, D.-S. (2005). Analyzing Dynamic Behavior of Large-Scale Systems through Model Transformation. International Journal of Software Engineering and Knowledge Engineering, 15(01), 35-60.

Shinkawa, Y. (2006). Inter-model consistency in uml based on cpn formalism. Paper presented at the 13th Asia Pacific Software Engineering Conference. APSEC 2006. .

Simmonds, J., & Bastarrica, M. C. (2005). A tool for automatic UML model consistency checking. Paper presented at the Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.

Sommervile, l. (2007). Software Engineering: 8th Edition, Addison Wesley.

Sommerville, I. (2011). Software Engineering: 9th edition. Pearson.

Spanoudakis, G., & Zisman, A. (2001). Inconsistency management in software engineering: Survey and open research issues. Handbook of software engineering and knowledge engineering, 1, 329-380.

Staines, T. S. (2008). Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept Petri net diagrams and colored Petri nets. Paper presented at the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. ECBS 2008. .

Stemberger, M. I., Kovacic, A., & Jaklic, J. (2007). A methodology for increasing business process maturity in public sector. Interdisciplinary Journal of Information, Knowledge, and Management, 2, 119-133.

Stephan, M., & Cordy, J. R. (2013). A survey of model comparison approaches and applications. Modelsward, 265-277.

Sun, P., & Jiang, C. (2009). Analysis of workflow dynamic changes based on Petri net. Information and Software Technology, 51(2), 284-292.

Sun, X., Li, B., Tao, C., Wen, W., & Zhang, S. (2010). Change impact analysis based on a taxonomy of change types. Paper presented at the IEEE 34th Annual Computer Software and Applications Conference (COMPSAC).

Tadj, C., & Laroussi, T. (2005). Dynamic verification of an Object-Rule knowledge base using Colored Petri Nets. Journal of Systemics, Cybernetics and Informatics, 4(3), 23-31.

Tam, T., Greenberg, S., & Maurer, F. (2000). Change management. Paper presented at the Western Computer Graphics Symposium, Panorama Mountain Village, BC, Canada.

Tang, Z. (2002). Temporal logic programming and software engineering. Bering: Science Press, vcI1.(1, 2), 5.

TGIgroup. (2013). Welcome to the Petri Nets World. Retrieved 2010, from http://www.informatik.uni-hamburg.de/TGI/PetriNets/

Thomas, O., Dollmann, T., & Loos, P. (2007). Towards Enhanced Business Process Models Based on Fuzzy Attributes and Rules. Paper presented at the Proceedings of the 13th Americas Conference on Information Systems: August 09–12, Keystone, Colorado, USA.

Torchiano, M., & Ricca, F. (2010). Impact analysis by means of unstructured knowledge in the context of bug repositories. Paper presented at the Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.

Torre, D. (2015). On validating UML consistency rules. Paper presented at the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW).

Torre, D., Labiche, Y., & Genero, M. (2014). UML consistency rules: a systematic mapping study. Paper presented at the Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering.

Trickovié, I. (2000). Formalizing activity diagram of UML by Petri nets. Novi Sad J. Math, 30(3), 161-171.

Tripathi, U. K., Hinkelmann, K., & Feldkamp, D. (2008). Life cycle for change management in business processes using semantic technologies. Journal of Computers, 3(1), 24-31.

Tsiolakis, A., & Ehrig, H. (2000). Consistency analysis of UML class and sequence diagrams using attributed graph grammars. Paper presented at the Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin.

Usman, M., Nadeem, A., Kim, T.-h., & Cho, E.-s. (2008). A survey of consistency checking techniques for uml models. Paper presented at the Advanced Software Engineering and Its Applications. ASEA 2008.

Van Der Aalst, W. (1999). How to handle dynamic change and capture management information? An approach based on generic workflow models. Computer Systems Science and Engineering, 16(5), 295-318.

Van Der Aalst, W. M. (2002). Inheritance of dynamic behaviour in UML. MOCA, 2, 105-120.

Van Der Straeten, R. (2005). Inconsistency Management in Model-Driven Engineering. PhD thesis, Vrije Universiteit Brussel.

Van Der Straeten, R., Mens, T., Simmonds, J., & Jonckers, V. (2003). Using description logic to maintain consistency between UML models «UML» 2003-The Unified Modeling Language. Modeling Languages and Applications (pp. 326-340): Springer.

Van Hee, K. M., Lomazova, I. A., Oanea, O., Serebrenik, A., Sidorova, N., & Voorhoeve, M. (2006). Nested nets for adaptive systems Petri Nets and Other Models of Concurrency-ICATPN 2006 (pp. 241-260): Springer.

Vandewoude, Y., & Berbers, Y. (2002). Run-time evolution for embedded component-oriented systems. Paper presented at the International Conference on Software Maintenance. Proceedings. .

Vasilecas, O., Dubauskaitė, R., & Rupnik, R. (2011). Consistency checking of UML business model. Technological and Economic Development of Economy, 17(1), 133-150.

Verkoulen, P. A. (1994). A Framework for Information Systems Design based on Object-Oriented Concepts and Petri Nets. Paper presented at the CAiSE Workshop, Utrecht, The Netherlands.

VisualParadigmCompany. (2011). Visual-Paradigm. Retrieved from www.visual-paradigm.com

Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation ECOOP 2007–Object-Oriented Programming (pp. 600-624): Springer.

Wagenhals, L. W., Haider, S., & Levis, A. H. (2002). Synthesizing executable models of object oriented architectures. Paper presented at the Proceedings of the conference on Application and theory of petri nets: formal methods in software engineering and defence systems-Volume 12.

Wagenhals, L. W., Haider, S., & Levis, A. H. (2003). Synthesizing executable models of object oriented architectures. Systems Engineering, 6(4), 266-300.

Wang, C. H., & Wang, F. J. (2007). An Object-Oriented Modular Petri Nets for Modeling Service Oriented Applications. Paper presented at the 31st Annual International Computer Software and Applications Conference. COMPSAC 2007. .

Watanabe, H., Tokuoka, H., Wu, W., & Saeki, M. (1998). A technique for analysing and testing object-oriented software using coloured Petri nets. Paper presented at the Asia Pacific Software Engineering Conference.

Weber, B., Rinderle, S., & Reichert, M. (2007). Change patterns and change support features in process-aware information systems. Paper presented at the Advanced Information Systems Engineering.

Weber, B., Sadiq, S., & Reichert, M. (2009). Beyond rigidity–dynamic process lifecycle support. Computer Science-Research and Development, 23(2), 47-65.

WebSPN-Research-Group. (2009). WebSPN 3.3 Web-accessible non Markovian Petri net tool Retrieved 2009, from https://mdslab.unime.it/webspn/

Weiser, M. (1984). Program slicing. Software Engineering, IEEE Transactions on(4), 352-357.

Weske, M. (1998). Flexible modeling and execution of workflow activities. Paper presented at the Proceedings of the Thirty-First Hawaii International Conference on System Sciences.

Westergaard, M. (2007). Behavioural Verification and Visualisation of Formal Models of Concurrent Systems. PhD dissertation.–Aarhus: University of Aarhus, 2007.–183 c.

Westergaard, M., & Verbeek, H. M. W. (2013). CPN Tools, from http://cpntools.org/

Williams, B. J., & Carver, J. C. (2010). Characterizing software architecture changes: A systematic review. Information and Software Technology, 52(1), 31-51.

Wolf, K. (2009). LoLA - A Low Level Petri Net Analyser. Retrieved from http://wwwteo.informatik.uni-rostock.de/ls_tpp/lola/

Wordsworth, J. B. (1999). Getting the best from formal methods. Information and Software Technology, 41(14), 1027-1032.

Wörzberger, R., Ehses, N., & Heer, T. (2008). Adding support for dynamics patterns to static business process management systems. Paper presented at the Software Composition.

Wuyts, R. (2001). A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. PhD thesis, Vrije Universiteit Brussel.

Xiaoning, F., Zhuo, W., & Guisheng, Y. (2008). Hierarchical Object-Oriented Petri Net Modeling Method Based on Ontology. Paper presented at the International Conference on Internet Computing in Science and Engineering. ICICSE'08. .

Xing, Z., & Stroulia, E. (2005). Analyzing the evolutionary history of the logical design of object-oriented software. Software Engineering, IEEE Transactions on, 31(10), 850-868.

Yang, S. J., & Chen, C.-C. (2003). An Integrated Approach for Workflow Process Modeling and Analysis Using UML and Petri Nets. MIS Review, 11, 47-75.

Yao, S., & Shatz, S. M. (2006). Consistency checking of UML dynamic models based on petri net techniques. Paper presented at the 15th International Conference on Computing, CIC'06. .

Yu, Z., & Cai, Y. (2006). Object-oriented Petri nets based architecture description language for multi-agent systems. IJCSNS, 6(1).

Zalewski, M., & Schupp, S. (2006). Change impact analysis for generic libraries. Paper presented at the 22nd IEEE International Conference on Software Maintenance. ICSM'06. .

Zamani, B., & Butler, G. (2013). Pattern Language Verification in Model Driven Design. Information Sciences, 237, 343-355.

Zapf, M., & Heinzl, A. (1999). Techniques for Integrating Petri-Nets and Object-Oriented Concepts: Working Papers in Information Systems, University of Bayreuth.

Zeng, L., Flaxer, D., Chang, H., & Jeng, J.-J. (2002). PLM flow—Dynamic Business Process Composition and Execution by Rule Inference Technologies for E-Services (pp. 141-150): Springer.

Zhao, J. (2002). Change impact analysis for aspect-oriented software evolution. Paper presented at the Proceedings of the International Workshop on Principles of Software Evolution.

Zhao, X., & Liu, C. (2007). Version management in the business process change context Business Process Management (pp. 198-213): Springer.

Zhao, Y., Fan, Y., Bai, X., Wang, Y., Cai, H., & Ding, W. (2004). Towards formal verification of UML diagrams based on graph transformation. Paper presented at the IEEE International Conference on E-Commerce Technology for Dynamic E-Business.

zur Muehlen, M., Indulska, M., & Kamp, G. (2007). Business process and business rule modeling: a representational analysis. Paper presented at the Eleventh International IEEE EDOC Conference Workshop. EDOC'07. .

Zur Muehlen, M., Indulska, M., & Kittel, K. (2008). Towards integrated modeling of business processes and business rules. Paper presented at the Proceedings of the 19h Australasian Conference on Information Systems (ACIS)-Creating the Future: Transforming Research into Practice, Christchurch, New Zealand.

LIST OF PUBLICATIONS AND PAPERS PRESENTED

Academic Journals

- Bassam Rajabi and Sai Peck Lee. 2017. Change Management Framework to Support UML Diagrams Changes. International Arab Journal of Information Technology. Accepted September 2017 (ISI/SCOPUS Indexed Publication)

- Rajabi, B.A. & Lee, S.P., (2017). Change Management Technique for supporting Object Oriented Diagrams Changes. *Computer Systems Science and Engineering*. (ISI-Indexed ISSN 0267-6192)

- Rajabi, B.A. & Lee, S.P., (2013). Consistent Integration between Object Oriented and Coloured Petri Nets Models. *The International Arab Journal of Information Technology Volume 11(4).* (ISI Indexed ISSN: 1683-3198)

- Rajabi, B., & Lee, S. P. (2010). Modelling and Analysis of Change Management in Dynamic Business Process. *International Journal of Computer and Electrical Engineering (IJCEE), Volume 2, Number 1*, 199-206. (Indexed by EI INSPEC ISSN: 1793-8198)

- Rajabi, B., & Lee, S. P. (2009b). Change Management in Business Process Modelling Based on Object Oriented Petri Net. *International Journal of Business, Economics, Finance and Management Sciences, Volume 1 Number 1 2009*, 72-77. (Indexed by Scopus ISSN:2073-0519)

- Rajabi, B., & Lee, S. P. (2009e). A Study of the Software Tools Capabilities in Translating UML Models to PN Models. *International Journal of Intelligent Information Technology Application (IJIITA), Volume 2 (5)*, 224-228. (Indexed by EI INSPEC, Zentralblatt MATH,CAB Abstracts, and EBSCO)

Conference Proceedings

- Rajabi, B., & Lee, S. P. (2009a). Change Management in Business Process Modelling Based on Object Oriented Petri Net. *Proceedings of World Academy of Science, Engineering and Technology, Volume 38*, pp. 12-17. Penang, Malaysia. (Indexed by Scopus ISSN:2070-3740)

- Rajabi, B., & Lee, S. P. (2009c). Runtime Change Management Based on Object Oriented Petri Net. *International Conference on Information Management and Engineering (ICIME 2009). 13*, pp. 42-46. Kuala Lumpur, Malaysia: IEEE Computer Society. (ISI Indexed ISBN:978-1-4244-3774-0)

- Rajabi, B., & Lee, S. P. (2009d). Change Management in Business Process Modelling Survey. *International Conference on Information Management and Engineering (ICIME 2009). 13*, pp. 37-41. Kuala Lumpur,Malaysia: IEEE Computer Society. (ISI Indexed ISBN:978-1-4244-3774-0)