computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

COEVOLUTION PATTERNS TO DETECT AND MANAGE UML DIAGRAMS CHANGES

Bassam Atieh Rajabi, Sai Peck Lee

Software Engineering Department, Faculty of Computer Science & Information Technology,
University of Malaya, Malaysia, bassamrajabi@gmail.com, saipeck@um.edu.my

Paper history:
Received 22 January 2019
Received in revised form 03 July 2019
Accepted 02 December 2019
Available online 31 December 2019

Keywords:
Coevolution;
Patterns;
UML;
Coloured Petri Nets;
Software Change.

Abstract: UML 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. 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 UML diagrams. In this research, a set of 84 coevolution patterns for supporting coevolution among UML diagrams are proposed to trace the diagrams' inconsistencies and to determine the change impact incrementally after updating diagrams elements. Coevolution patterns are applied to UML class, object, activity, statechart, and sequence diagrams to cover the different perspectives of UML diagrams. The researcher uses CPNs as a formal language of modelling case study models for the proposed patterns. CPNs tools simulation and monitoring toolboxes are used to validate and monitor the proposed coevolution patterns models and to collect quantitative data about the patterns.

1. INTRODUCTION

Software engineers continue to face challenges in designing adaptive and flexible software systems that can cope with requirements change. One of the crucial challenges in software change management is to preserve the coevolution among software system artefacts. Understanding the coevolution, which represents the dependency between artefacts that frequently change together. Coevolution involves both change impact analysis and change propagation between software artefacts or models, and hence, it is required to check if the change in one of the artefacts ultimately affects the other artefacts and to maintain the consistency between artefacts. For an efficient coevolution check, change impact analysis is an important step to analyse and determine the change effect, identify the parts that require retesting, and maintain the consistency among software artefacts. Identifying all components affected by the change is based on the traceability analysis to analyse the dependencies between and across software artefacts at all levels of the software process. 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. 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); each category focuses on modelling a different perspective of a problem domain. One of the critical issues is to preserve the coevolution among these diagrams so that they can be updated continuously to reflect software changes. UML is

the de-facto standard for modelling OO software systems. UML defines different diagrams. Relations between these diagrams are complex, and may lead to inconsistent UML diagrams. Coevolution among different perspectives or views of UML diagrams means that the modification in one diagram should be reflected in other related diagrams to ensure the consistency of all diagrams. 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. 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 patterns for supporting coevolution between UML diagrams. The proposed patterns aim 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. Additionally, a change history between two versions created from the same diagram is addressed.

The concept of pattern was introduced to expresses a relation between a certain context, a problem, and a solution. Design patterns in OO design capture frequently recurring sub-designs or groups of objects that collaborate to perform a certain task [1, 2]. The researcher studied the state of the art patterns mainly patterns proposed by Gamma's [1, 2] and proposed a new set of patterns to support coevolution between UML diagrams including change impact and traceability analysis. 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, Coevolution patterns are applied on Class Diagram (CD), Object Diagram (OD), Activity Diagram (AD), StateChart Diagram (SCD), and Sequence Diagrams (SD). These diagrams cover the three perspectives of UML diagrams (i.e. structural, behavioural, and interaction). Several studies mentioned that these diagrams are the mostly used diagrams in UML design. Additional patterns for change control and management are also provided. The relations between these patterns are identified and stated clearly. UML is a powerful means for describing the static and dynamic aspects of systems, but remains semi-formal and lacks techniques for model validation and verification. Formal specifications and mathematical foundations such as CPNs are widely used in handling of inconsistency problems among models to automatically validate and verify the model dynamic behaviour. In this research, CPNs Tools are used to creates, simulates, and validates the proposed patterns. Previous approaches are concentrated 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. The proposed patterns design handle the coevolution between UML diagrams perspectives and ensuring the consistency and coevolution of all diagrams comprehensively. The proposed pattern design enables comprehensive modelling for changes in UML diagrams and provides coevolution patterns for all type of change including the change impact and traceability analysis for UML diagram changes (i.e. it improves pattern support in software analysis and design). Additionally, it provides a new structure for the CPNs to support model changes and it increases the structuring capabilities of CPNs. This section introduces the research context. The rest of the paper is organized as follows: Section 2 presents a literature review for this research. In section 3, the coevolution patterns to support detecting and resolving the coevolution and inconsistencies among UML diagrams are proposed. Section 4 is dedicated to the proposed patterns analysis and results discussion. Finally, conclusions are drawn and suggested recommendations for some potential future research areas are highlighted.

2. LITERATURE REVIEW

Pattern languages 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 [3]. 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 [1] and Gamma, Helm, Johnson, and Vlissides [2]). Gamma, et al [1] and Gamma, et al [2] proposed a pattern definition for use in OO software design. This pattern is defined as follows: *Intent, Motivation, Applicability, Participants, Collaborations, Diagram, Consequences, Implementation, Example, and See*

Also. Patterns are used in many workflow software systems to manage and execute operational processes involving people, applications, and/or information sources. Some of these patterns are modelled and simulated using CPNs. Pattern language verification in the model-driven design approach is introduced in [4]. A pattern language for evolution reuse in component-based software architectures approach is proposed in [5]. 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 [6]. The main ideas and weaknesses of these approaches are: Standard Object Constraints Language (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 [7],[8]. Some approaches that use OCL to ensure consistency between UML diagrams are proposed in Egyed [9], [10] and Elaasar and Briand [11]. 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 [12]. 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 [13]. According to Lucas et al. [14], 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 CPNs. 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. A coevolution approach between a component-based architecture model and OO source code is proposed in Langhammer [15]. García, Diaz, and Azanza [16] 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 [17], model coevolution describes the problem of adapting models when their metamodels evolve. Other approaches in consistency and coevolution based on transformational models are presented in other studies [18-20]. 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. Improving the effectiveness and the accuracy of state-of-the-art coevolution techniques in managing UML 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. Previous approaches are concentrated 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 coevolution patterns to cover these limitations. A formal modelling language based on CPNs is used to model and simulate the proposed patterns. 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.

3. PROPOSED COEVOLUTION PATTERNS

In this research, coevolution patterns are proposed in order to provide a systematic and methodical approach for managing changes among UML structural, behavioural, and interaction diagrams. The proposed patterns are 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. 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, information about change impact and traceability analysis are identified for all types of change 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. The proposed coevolution patterns are identified and categorized based on UML diagram categories and relations (structural, behavioural, and interaction diagrams). The formal approach is used to model, simulate, and validate the proposed coevolution patterns using the CPNs formal modelling tool. The steps of defining the proposed patterns are shown in Figure 1 and discussed in details in the following sub-sections. Design of consistency rules and the methods and algorithms in determining the components affected by a change are proposed in the researcher previouse work provided in [21]. The consistency 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. 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.

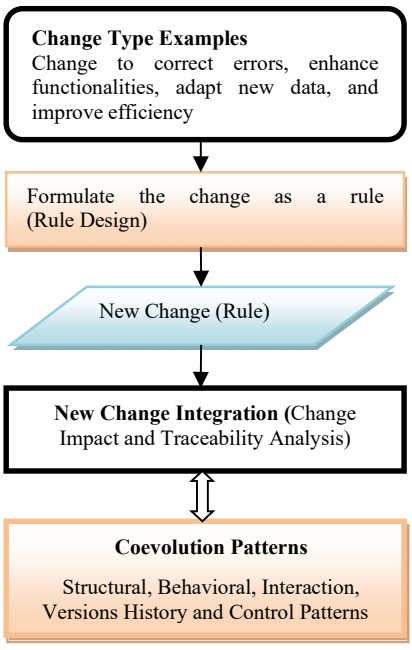

Figure 1 – Steps of Defining Proposed Coevolution Patterns

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. 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 based on the formal definitions provided in [21]. Theses formal definitions are used to find the impact-related elements, reflexive relation, transitive relation, relation between UML Diagram elements and change types, and the relation between UML Diagram versions.

3.1 PROPOSED COEVOLUTION PATTERNS

Generaly, developers have focused on using patterns in software modelling as design patterns and in the workflow software management system. In comparing with other approaches, previous approaches are concentrated 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 and coevolution of all diagrams comprehensively. In this research, a new pattern design for the coevolution between UML diagrams is suggested. The proposed pattern design includes the change impact and traceability analysis information. In this research, coevolution patterns are identified and categorized based on UML diagrams categories and relations. 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. The proposed patterns modifies Gamma , et al [1] and Gamma , et al [2] patterns 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 helps 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 CPN modelling techniques.*

Change impact and traceability analysis*:*

The proposed impact and traceability analysis information 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.

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 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 following are 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 proposed patterns design includes information about the change impact and traceability analysis. In order to include this information in the patterns design and simulation, a new structure for the mutual integration of UML and CPNs modelling languages is proposed in [22, 23] to support the coevolution between UML diagrams and for framework modelling and simulation. In the proposed structure, consistency and integrity rules are part of the transformation process and integrated in the transformed CPNs model. The consistency rules include a set of rules to check and maintain the consistency and integrity based on the relations between UML diagrams. The proposed OOCPN structure is defined by the tuple n = (∑, Pg, P, Fp, T, SubT, A, N, C, G, E, M0, R), where:

∑: is a finite set of non-empty types, called colour sets
Pg: {Pg0, Pg1….Pgn} is a set of pages, where Pg0 is the main page
P: is a finite set of places
Fp: is a finite set of fusion places
T: is a finite set of transitions
SubT: is a finite set of substitution transitions
A: represents a set of directed arcs
N: is a node function
C: is a colour function
G: is a guard function
E: is an arc expression function
M0: P → C is the initial (coloured) marking
R: is a finite set of consistency and integrity rules

A. Proposed Class Diagram Patterns

Create an element: class, attribute, operation, class inheritance, association relationship, aggregation relationship, composition relationship. Modify an element: class name, attribute name, attribute visibility, attribute property,attribute type, attribute value**,** operation property, operation type, operation visibility, operation name, generalization relationship, association destination multiplicity, association source multiplicity, role name. Delete an element: class, attribute, operation, generalization relationship, association relationship, aggregation relationship, composition relationship. Search about an element: class, attribute, operation, generalization relationship, association relationship, aggregation relationship, and composition relationship. 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.

B. Proposed Object Diagram Patterns

Create an element: message data type, variable/ message. Modify an element: object name, message data type, variable/message. Delete an element: object, variable/message. Search about an element: instance name, object exist, instance class. Consistency check: check object name, objects not created.

C. Proposed Activity Diagram Patterns

Create an element: activity, a sub-activity, control node, action, iteration, guard condition. Modify an element: sub-activity, control node, action, iteration, guard condition. Delete an element: activity, sub-activity, control node, action, iteration, guard condition. Search about an element: activity, sub-activity, action, fork, join, decision, merge, object, loop, guard, call behaviour action.

Consistency check: objects not in Ads, ADs not created, AD elements not created, modify AD name

D. Proposed Statechart Diagram Patterns

Create an element: start/end node, event, state, action, iteration, guard condition. Modify an element: event, action, iteration, guard condition. Delete an element: event, start/end node, action, iteration, guard condition. Search about an element: event, action, guard, loop. Consistency check: SCDs not created ,SCD elements not created, modify SCD name.

E. Proposed Sequence Diagram Patterns

Create/Modify/Delete an element: object, message, Create/ Delete/ Modify iteration, Create/Delete/Modify guard condition, Create/Delete/Modify operators. Search about an element: object, message, loop, guard, Opt, Ref, Alt, Par. Consistency check: SDs not created, SD search, SD elements not created, objects not in SDs, Modify SD name patterns.

F. Proposed Change Control Patterns

Search Patterns: find a diagram element. Class Diagram Search Patterns: find a class diagram element. Object Diagram Search Patterns: find an object diagram element. Activity Diagram Search Patterns: find an activity diagram element. Sequence Diagram Search Patterns: find a sequence diagram element. Change History Patterns: Changes history selection, Store in file. Update new version

The following is an example about creating new operation pattern.

CD Create New Operation Patterns

Intent: To maintain the coevolution between diagrams based on the change 'creating new operation'.

Motivation: The adding of new operations is mandatory in any class update. Maintaining the coevolution of the class diagram after this update operation is important.

Problem description: To add a new operation, a set of consistency rules should be maintained before applying the change. Additionally, checking the redundancy of the operation names is important as discussed in relation to **Помилка! Джерело посилання не знайдено.**.

Solution/Diagram: This pattern is solves the above problem by performing the following steps:

1. Applying **Помилка! Джерело посилання не знайдено.** to check the redundancy of the new operation name and the consistency rules before adding the operation.
2. If the result of Step 1 is 'this operation name exists' the new operation name will be rejected.
3. If the result of Step 1 shows that the operation name is unique and the consistency rules are checked, then the change will be made.
4. The new operation will be added to the list of class diagram operations.
5. The new operation will be added to the list operations not created in the activity, statechart, and sequence diagrams in order to maintain the consistency between diagrams.
6. The new changes will be stored in a file for change history management.

Figure 2 shows the solution diagram.

Change impact and traceability analysis: *Change Type:* **Create a new operation**

Change Impact: GC *Affected Diagrams*: All

Consistency and Integrity Rules:

If (a new operation is created) Then (No private/protected attribute or operation can be accessed by an operation of another class)

If (an operation has a pre or post condition attribute) Then ((All diagrams' attributes/operations must be defined in the CD) ∩ (attribute type must be compatible))

If (an operation realizes an interface operation) Then

- (Its ((owner scope values) ∩ (polymorphic properties) ∩ (precondition) ∩ (concurrency values) ∩ (query properties)) must be the same as that of the interface operation)
- (The directions of all the parameters must match the directions of the parameters of the interface operation)

If (an attribute changeability is not "changeable") Then (A diagram element cannot update an attribute if the attribute changeability is not "changeable").

Example: 1`(["Class1"],"Op1").

Related patterns:

Pattern 2. Operation Redundancy Check Pattern.
Pattern 4. Class with No Operation or Attribute Consistency Check.
Pattern 5. Class Element Redundancy Check
Pattern 9. Class Diagram Operation Search
Pattern 19. Activity Diagrams Not Created
Pattern 20. Activity Search
Pattern 22. Activity Diagram Elements Not Created
Pattern 23. Activity Diagram Action Search
Pattern 33. Sequence Diagram Not Created
Pattern 34. Sequence Diagram Search
Pattern 49. Class Diagram Create New Operation
Pattern 59. Class Diagram Delete Operation

Pattern 70. Class Diagram Modify Operation Property
Pattern 71. Class Diagram Modify Operation Type
Pattern 72. Class Diagram Modify Operation Visibility
Pattern 73. Modify Sequence Diagram Name
Pattern 74. Modify Operation Name
Pattern 78. Modify Activity Diagram Name

3.2 CASE STUDY MODELS

Case study models are modelled for the class, object, activity, statechart, and sequence diagram.. 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:** The CD elements that are modelled in CPNs are *attributes, values, operations, classes, abstract classes, communication methods and dynamic binding, generalization/class inheritance, associations, aggregation, composition, navigability arrow, polymorphism, multiplicity, role name, an interface, and dependency.* **Object Diagram:** The OD that are modelled in CPNs are *object (class instance), and object state.* **Activity Diagram:** The AD 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:** The SD 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 SCD 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.

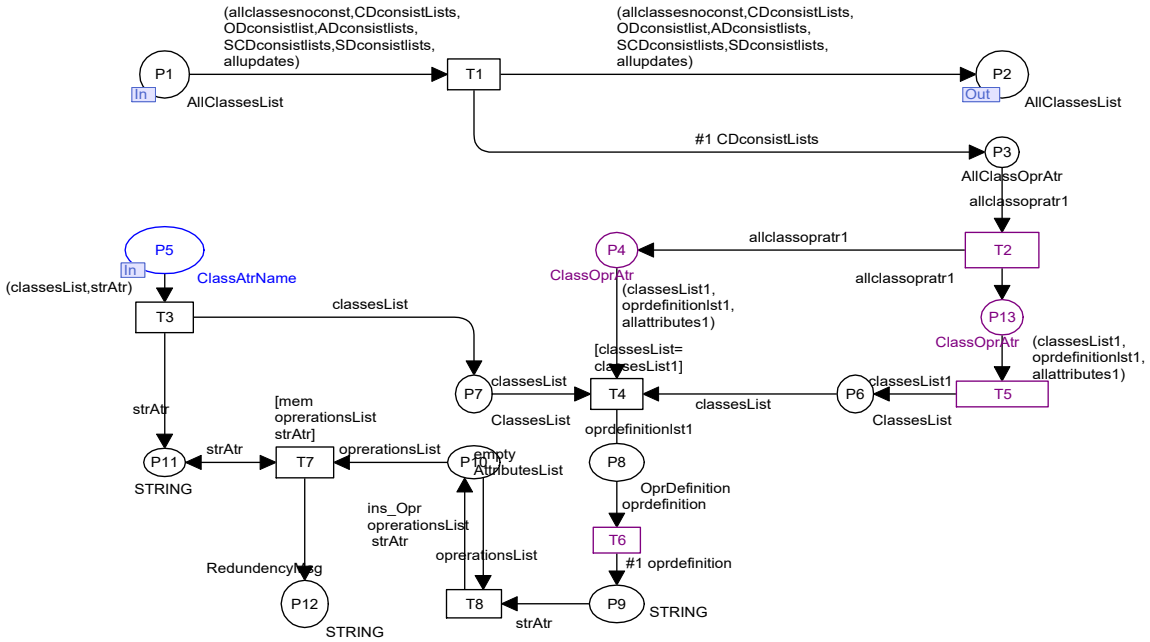

Figure 2 – CD Create New Operation

3.3 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 following stages: designing the pattern diagram, running the simulation, and 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. 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.

A. Validation threats:

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). 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 patterns, the simulation starts with the diagram simulation. 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 1 summarizes the simulation steps needed for the case study models.

Table 1. 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

4. ANALYSIS AND DISCUSSION

In this section, the performance of the proposed patterns is analysed and discussed also compared with the state-of-the-art.

4.1 CHANGE IMPACT AND TRACEABILITY ANALYSIS 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. 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 ≥ 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 impact on the diagrams and elements, the more severe the change. The results show that the relation between the class diagram and other models is strong. This explains number of 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* ∈ AD: If (*e* is changed) Then (SCD and SD are affected).
- ∃ *e* ∈ SCD: If (*e* is changed) Then (AD and SD are affected) *The dynamic behaviour of the SCD is described using the AD, SD.*
- ∃ *e* ∈ SD: If (*e* is changed) Then (AD and SCD are affected)

The number of update operations supported for each diagram is provided in Figure 3. Self, direct, and indirect dependencies are considered. In comparison with the approaches it is not check only the consistency between two versions from the same diagram.

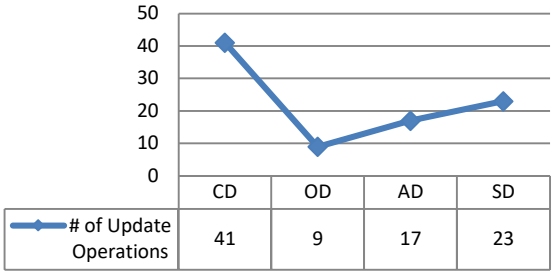

Figure 3 – Number of Update Operations Supported for Each UML Diagram

4.2 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 main goal was to find a way to utilize 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 Figures 4 and 6. The proposed pattern design supports the automatic checking of consistency during the diagrams design process not just the checking the consistency of the diagrams when they are updated. This can be considered a major advantage over the state-of-the-art approaches. 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; 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. 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.

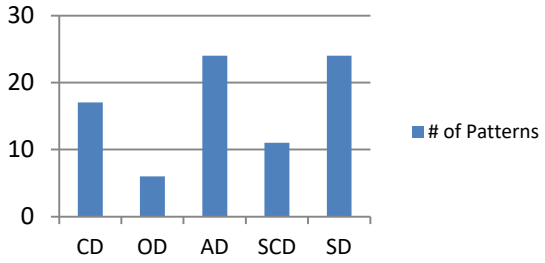

Figure 4 – Diagrams Patterns

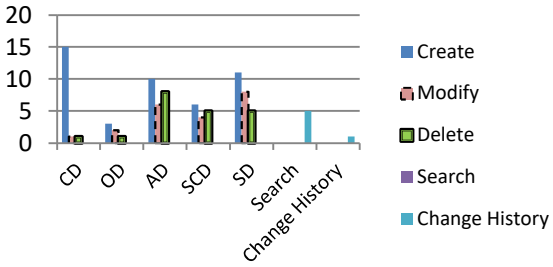

Figure 5 – Number of Proposed Patterns

4.3 VALIDATION AND PERFORMANCE ANALYSIS

The proposed patterns 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. Patterns Validation

The simulation capabilities of CPNs Tools are used to execute the patterns 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 patterns. 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 Figures 6 and 7.

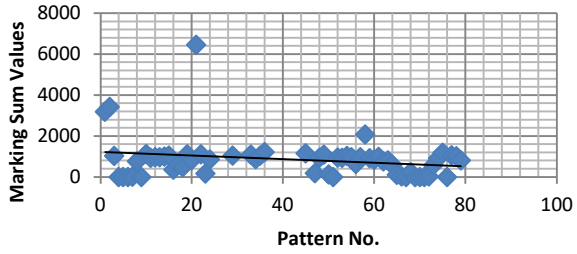

Figure 6 – Analysis of Patterns Marking Size Sum

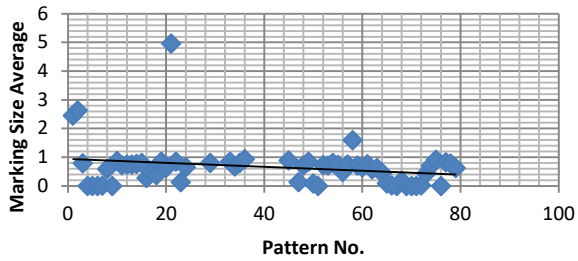

Figure 7 – Analysis of Patterns Marking Size Average

B. Data Collector and Marking Size Monitoring

Table 2 illustrates the proposed patterns 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 3 summarizes the marking size monitoring data and data analysis results. The average metrics are calculated by Sum/Count.

Table 2. The Model Elements in the Proposed Patterns Model

Diagram Element	Statistics (Number of Elements)
Places	2126
Place Instances	2274
Transitions	942
Transitions Instances	1418
Arcs	3638
Arcs Instances	4450
Pages	191
Pages Instances	267
Declaration)	262
Types	132
Variables	141

Table 3. Analysis of Marking Size Monitoring Data

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

4.4 DISCUSSION

In related works, patterns that are provided are specified only for modelling the business process and workflow software management systemand the patterns approach are used as design patterns. In contrast, the patterns proposed in this research can be used to deal with software changes in any OO diagrams design. According to [3], patterns exist not only as design patterns, but for every phase of software development, including requirements analysis, architectural design, implementation , and testing. The proposed patterns can also be applied to these phases in addition to the software maintenance phase. The proposed patterns produce 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 patterns reduces interdependencies between the model components, and facilitates easy maintenance and updates without impacting the entire model. The proposed patterns are not a comparison between two versions only.
- Maintainability: Enhances the diagrams' change support through building a consistent model at the design time, and then, applying the changes to these 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 models (Incremental and Automatic correctness check using CPNs simulation and monitoring tools).
- Integrity: Integrate the new changes with the current diagrams.

The main limitations of this research are as follows: The proposed patterns are restricted on term of the range of UML diagrams supported in the patterns design (specifically class, object, activity, statechart, and sequence diagrams). Hence more comprehensive patterns are required to cover all diagrams. 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.

5. SUMMARY AND FUTURE WORK

In this research, a novel approach for coevolution patterns were proposed to manipulate the change effect in the UML diagrams' elements. The proposed patterns can be applied to detect the diagram elements affected by a change in a system design modelled using UML diagrams. These patterns 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. The proposed coevolution patterns support the UML class, object, activity, statechart, and sequence diagrams because the coevolution between these diagrams is very high. 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, The work done in this thesis could be extended in several directions: The proposed patterns cover some of the UML diagrams, more comprehensive patterns could be attempted in a future research study. Additionally, extending the research by considering the semantic meanings of the model and considering the coevolution between models and the source code.[5]

6. REFERENCES

[1] E. Gamma, et al., *Design Patterns: Rlements of Reusable Object Oriented Software*,: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[2] E. Gamma, et al., *Design patterns: Abstraction and Reuse of Object-oriented Design*, Springer, 2001.

[3] I. Côté, and M. Heisel, "Supporting evolution by models, components, and patterns," Proceedings of the 1 Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): "Design for Future – Langlebige Softwaresysteme", 2009, pp. 39-51.

[4] B. Zamani, and G. Butler, "Pattern language verification in model driven design," *Information Sciences*, vol. 237, pp. 343-355, 2013.

[5] A.A. Abbasi, A Pattern Language for Evolution Reuse in Component-based Software Architectures," Dublin City University, 2015.

[6] P. Sapna, and H. Mohanty, "Ensuring consistency in relational repository of UML models," *Proceedings of the 10th IEEE International Conference on Information Technology, (ICIT'2007)*, 2007, pp. 217-222.

[7] S. Lehnert, A Review of Software Change Impact Analysis, Ilmenau University of Technology, Tech. Rep, 2011.

[8] A. Khalil, and J. Dingel, *Supporting the Evolution of UML Models in Model Driven Software Development: A Survey*, School of Computing, Queen's University, Technical Report 2013-602, 2013.

[9] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, issue 2, pp. 188-204, 2011.

[10] A. Egyed, et al., "Maintaining consistency across engineering artifacts," *Computer*, vol. 51, issue 2, pp. 28-35, 2018.

[11] M. Elaasar, and L. Briand, *An Overview of UML Consistency Management*, Carleton University, Canada, Technical Report SCE-04-18, 2004.

[12] A. Sharaff, "A methodology for validation of OCL constraints using coloured Petri nets," *International Journal of Scientific & Engineering Research*, vol. 4, no. 1, pp. 1-6, 2013.

[13] J.P. Puissant, *Resolving Inconsistencies in Model-Driven Engineering using Automated Planning*, PhD thesis, University de Mons, 2012.

[14] F.J. Lucas, F. Molina, and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, issue 12, pp. 1631-1645, 2009.

[15] M. Langhammer, "Co-evolution of component-based architecture-model and object-oriented source code," *Proceedings of the 18th International Doctoral Symposium on Components and Architecture*, 2013, pp. 37-42.

[16] J. García, O. Diaz, and M. Azanza, "Model transformation co-evolution: A semi-automatic approach," *Software Language Engineering*, Springer, 2013, pp. 144-163.

[17] Z. Protic, *Configuration Management for Models: Generic Methods for Model Comparison and Model Co-evolution*, PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2011.

[18] A. Kusel, et al., "Systematic co-evolution of OCL expressions," Proceedings of the 11th Asia-Pacific Conference on Conceptual Modelling (APCCM 2015), Sydney, Australia, 27-30 January 2015, pp. 33-42.

[19] A. Demuth, et al., "Co-evolution of metamodels and models through consistent change propagation," *Journal of Systems and Software*, vol. 111, pp. 281-297, 2016.

[20] D. Torre, et al., "A systematic identification of consistency rules for UML diagrams," *Journal of Systems and Software*, vol. 144, pp. 121-142, 2018.

[21] B.A. Rajabi, and S.P. Lee, "Change management technique for supporting object oriented diagrams changes," *Computer Systems*

Science and Engineering, vol. 32, no. 1, pp. 49-63, 2017.

[22] B.A. Rajabi, and S.P. Lee, "Consistent integration between object oriented and coloured Petri nets models," *The International Arab Journal of Information Technology*, vol. 11, issue 4, pp. 406-415, 2014.

[23] B.A. Rajabi, and L. Sai Peck, "Change management framework to support UML diagrams changes," *The International Arab Journal of Information Technology*, vol. 16, issue 4, pp. 720-730, 2019.

Bassam Rajabi *received his PhD degree in Software Engineering from University of Malaya, Malaysia. Currently, he is the dean of Ibrahimieh Community College. His areas of interest are Software Design and Modeling Techniques.*

Sai Peck Lee *is a Professor at University of Malaya. She obtained her Ph.D. degree in Computer Science from Université Paris 1 Panthéon-Sorbonne. Her current research interests include Object-Oriented Techniques and CASE tools, Software Reuse, Requirements Engineering and software quality.*