SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V13I4P102

Volume 13 Issue 4, 12-30, April 2026

Original Article

Structural Analysis and Approach of Software Evolution Process Applying Petri Nets

Rajeeb Sankar Bal[1], Jibendu Kumar Mantri[2]

[1,2]*Department of Computer Application, MSCB University, Odisha, India.*

[1]*Corresponding Author : rajiv.s.bal@gmai.com*

Received: 22 February 2026 Revised: 29 March 2026 Accepted: 13 April 2026 Published: 27 April 2026

Abstract - *In software engineering, software evolution is applied after the initial version of the software has been developed. Also, software evolution is based on the enhancement of the software system by adding new functionalities and correcting defects. Since such updates are complex and prone to errors, systematic methods are required to manage software changes successfully. Software process tailoring means reshaping standard organizational processes to light on project-specific needs while still maintaining compliance and needed verification and validation tasks or activities. Applying traditional process designs and standards supports better quality, decreases risks, and reduces rework in the development of software. In this paper, we consider software evolution processes modeled as basic blocks applied to Petri Nets (PNs) and tailored by four important operations: addition, deletion, splitting, and merging to modify workflows. In tailoring the software evolution process, each basic block is studied using control vectors, incidence matrices, and state equations, with reachability analysis ensuring the required process outcomes. The marked graph PNs represent structured concurrency without uncertainty and conflict, and confluence matrices support structural study by identifying competing and convergent execution paths, thereby protecting correctness and consistency of workflow behavior. Structural analysis shows basic blocks with various executions, i.e., sequence block preserves deterministic control flow, concurrency block represents parallel execution, selection block enforces valid choice paths, and iteration block identifies infeasible loops. According to the diagonal elements of the Conflict and Confluence matrices, one guarantees each activity is consistent and deterministic, allowing safe composition of basic blocks. Together, these results confirm that the Petri Net (PN) based application produces sound, reliable, and feasible software evolution processes. Effectively, PNs' basic blocks model in software evolution, and their matrix representations enable systematic analysis. Hence, Software processes can be tailored by main operations: addition, deletion, splitting, and merging, with consistency between high- and low-level models maintained to ensure correctness and reliability.*

Keywords - *Software Development, Software Evolution, Tailoring, Basic Blocks, Petri Nets.*

1. Introduction

Software evolution refers to the continuing modification and tailoring of software systems to make sure they remain useful, reliable, and aligned with changing user needs and technological environments. As software systems grow in scale and complexity, evolution becomes a critical point of long-term software sustainability. Software evolution plays an important role in development, as programmers infrequently build software from scratch and instead spend important effort modifying existing systems to add new features and fix defects. Evolving software systems is frequently time-consuming and error-prone, making maintenance, upgrading, migration, and systematic evolution key weightage in software engineering [1]. Lehman's influence on software evolution, developed with collaborators like Laszlo Belady, highlights that software systems cannot be completely specified for real-world use because both the environment and the software itself change over time. As programs are applied, they influence their environment, creating a response loop that drives further evolution. Even if software initially meets user requirements, it must be continuously changed as needs and conditions evolve. According to these observations, Lehman's Laws of Software Evolution assign one of the earliest theoretical foundations for understanding this process, explaining that long-lived systems require ongoing change, tend to increase, and may degrade in quality unless actively preserved. A detailed outline of these laws is presented in the accompanying Table 1. Modern software development progressively departs from traditional waterfall-style approaches, favouring agile processes and extensive application of open-source components. Developers frequently contribute to and adapt existing open-source codebases, creating specialized versions for specific requirements while also benefiting the broader community. Examples include the Android platform built on Linux and browsers like Chrome and Safari, which apply the WebKit

engine. These practices raise questions about how to estimate individual contributions and measure software characteristics when much of the code is modified rather than newly developed. Analyzing software evolution, productivity, and quality in these factors requires new models that reflect the realities of agile development and open-source reuse [2]. While early software lifecycle models, such as the waterfall model, positioned maintenance as a different phase following development, subsequent research has shown that evolution is an essential and unavoidable point of a system's entire lifespan. Modern development methodologies, including iterative, agile, and DevOps practices, further point up continuous change, rapid delivery cycles, and frequent stakeholder response. As a result, software evolution is no longer perceived as post-deployment maintenance but as a continuous, integrated activity. However, despite decades of analysis, existing evolution models and theories were essentially developed in contexts where requirements changed slowly, and system architecture was relatively stable. These traditional frameworks do not entirely occupy the dynamic, high-velocity environments in which contemporary software is developed. This mismatch presents various challenges: difficulty maintaining quality in continuously evolving systems, bounded guidance on how classical evolution principles apply to agile and DevOps workflows, and insufficient conceptual simplicity around how maintenance and evolution interconnect in modern practice [3-4]. Software evolution is the advancing process of adapting and maintaining systems to meet changing requirements and environments, a challenge underlined by Lehman's Laws, which show that long-lived software needs continuous modification to intercept quality degradation. Modern development progressively relies on agile methods and open-source components, where developers modify and extend existing code, raising challenges in evaluating contributions and ensuring reliable behavior. Traditional modeling with PNs and the use of basic blocks as structured, self-contained workflow units enable systematic analysis and controlled tailoring of evolution processes through addition, deletion, splitting, and merging operations. This approach ensures consistency, correctness, and predictability, supporting sound, modular, and reusable software evolution workflows.

Contribution: Software evolution is the continuous process of modifying and maintaining systems to meet changing requirements, a challenge underlined by Lehman's Laws, which show that long-lived software requires ongoing modification to prevent quality degradation. Modern development progressively relies on agile methods and open-source components, where developers modify and extend existing code, creating challenges in evaluating contributions and ensuring reliable behavior. Based on this, formal modeling with PNs and basic blocks as structured, self-contained workflow units enables systematic analysis and controlled tailoring of evolution processes by addition, deletion, splitting, and merging operations, ensuring consistency, correctness, and modularity. In this paper, the authors contribute by explaining key concepts of software evolution, examining the limitations of traditional models in modern development contexts, and proposing a conceptual framework that integrates Lehman's Laws with contemporary practices, advancing guidance for managing evolving systems and connecting foundational theory with practical engineering requirements.

Table 1. The Lehman's Laws of Software Evolution.

Laws	Description
Continuing Change	Software must continuously evolve to remain useful; otherwise, it becomes outdated or less satisfactory.
Self-Regulation	Software evolution is a self-regulating process, maintaining stability through internal adjustments.
Conservation of Organizational	Stability Over time, the rate of work (e.g., development activity) tends to stay stable, despite growth in system size.
Conservation of Familiarity	Developers and stakeholders need to retain a consistent understanding of the system, which limits how much it can change at once.
Continuing Growth	Software functionality must grow continually to meet evolving user needs and stay relevant.
Declining Quality	Without regular updates and improvements, software quality will decline over time due to environmental changes.
Feedback System	Software evolution relies on complex, multi-level feedback loops between developers, users, and the system.

2. Related Work

In [5], a rule-based specification approach for software process models is presented. Applying the German Process Model (GV-Model) as a case study and identifying multiple inconsistencies within its informal description highlights the need for a more precise and formalized representation. We have studied and aimed to correct these deficiencies and ensure that the GV-Model can be executed reliably. Also, a set of tailoring rules is proposed that allows organizations to derive customized and executable process models from standard models. This contribution is significant because it demonstrates how rule-based specifications can improve both the accuracy and adaptability of software process models. We have studied the transformation of PNs into graph-based representations. In [6], it is focused primarily on simple or basic PNs, largely because only nets with a finite number of

states or finite structural properties can be reliably converted into graphs. According to convertibility, PNs must typically be bounded or structurally constrained. Although it is possible to decrease complex PNs prior to conversion, such reductions often result in the loss of structural or behavioral detail. The marking-graph approach offers fewer restrictions, but it is susceptible to the well-known state explosion problem, particularly for large-scale models. These limitations have motivated the use of alternative techniques for analyzing and visualizing PN structures. Structural reduction rules, for example, can be applied in advance to simplify complex nets before generating their graph representations. Additionally, other forms of graphical modeling have been proposed as complementary or substitute approaches. In [7], a method for the automated analysis of system scenarios using PNs is presented. This approach enables the systematic evaluation of key behavioral properties such as boundedness and liveness, which directly support the assessment of correctness, consistency, and completeness in scenario-based PN models. By applying this method, issues related to these properties can be detected at early phases of software development, particularly when PNs are derived from scenario conditions. In addition, the method enhances traceability by linking identified PN level problems to their originating scenario descriptions, thereby facilitating clearer diagnosis and clarification of system requirements. In [8], a comprehensive review of foundational and advanced concepts in PN research is provided. The survey places significant emphasis on place/transition systems and core theoretical developments that suggest much of contemporary PN analysis. Also, it is highlighted that increasing research interest in extended PN formalisms, such as timed, stochastic, and high-level nets, describes how these models are being applied across an expanding range of domains. This body of work demonstrated the evolution of PN theory toward overflowing modeling capabilities and broader practical relevance. In [9], it is presented that the Sixth International Software Process Workshop (ISPW6) was held in Hakodate, Japan, from October 29 to 31, 1990. It was the sixth in a series of workshops devoted to discussing several aspects of software processes. The presented work included a combined model that represented the software development process as a sequence of distinct process steps executed in a continuously changing project environment. The viability of this model is demonstrated by integrating a detailed event model of the ISPW6 software development process with the dynamic structure of a system developed by Abdelhamid and Madnick. The collaboration of these two modeling approaches provided a foundation for examining issues and newly emerging problems that are particularly relevant to software project managers. In [10], the critical role of requirements in guiding decision-making during system development is discussed. Also, it is stated that, in the absence of well-defined requirements, it becomes difficult to determine which alternative activities or solutions should be pursued. Requirements act as a conceptual bridge between the problem domain and the implementation domain, ensuring that system objectives are accurately converted into technical specifications. In this sense, they serve as an essential interface between stakeholder expectations and developer interpretations, supporting clearer communication and more informed design choices. In [11], we have studied a computational framework that represents software evolution using a Three-Dimensional (3D) modeling structure. Also, it is stated that software evolution should be expressed in a 3D pattern that simultaneously captures the progression of software models, their underlying foundations, and their evolving states. The study proposed an initial approach for modeling these dimensions within a unified 3D space, offering an early step toward more comprehensive visualization and analysis techniques for software evolution. In [12], a structured methodology for tailoring software evolution processes is presented. The approach defined four fundamental customization operations: adding, deleting, splitting, and merging, used to core process building blocks. A notable contribution of this work is the emphasis on maintaining consistency between high-level and low-level process models during operations such as splitting and merging, thereby ensuring structural and semantic coherence. However, it is recognized that additional analysis is needed to assess the extent to which real-world software processes conform to or deviate from established reference models. This highlighted an ongoing challenge in bridging the gap between standardized process representations and practical software development practices. In [13], we have studied the effectiveness of PNs as formal models for capturing and analyzing concurrency within complex systems. The study introduced various PN matrix representations and emphasized their functionality in evaluating structural and behavioral properties of various PN subclasses. By leveraging matrix-based analysis techniques, the work highlighted new opportunities for systematic and computationally efficient examination of PN models. In [14], we have studied the software source code metrics that influence class complexity and emphasize the importance of early complexity identification to decrease testing and maintenance costs. Also, it is advocated that the use of complexity prediction models supports preventive quality management. The results showed that machine learning–based code quality prediction helps practitioners anticipate faults and upgrade overall software quality.

Gaps in existing studies: We have studied the existing rule-based approaches for software process models, considering correcting inconsistencies and enabling customization, but we do not plan systematic methods for modeling and analyzing dynamic process evolution. They have inefficient mechanisms to safely apply tailoring operations such as addition, deletion, splitting, and merging while ensuring consistency and correctness. Moreover, there is limited integration of formal modeling techniques, like PNs, with practical tailoring rules to support reusable and

analyzable software evolution processes [5]. Current PN-based outline analysis methods consider detecting boundedness and liveness issues within individual plans, but do not address the systematic modeling and evolution of complete software processes. They have inefficient mechanisms to evaluate the effect of tailoring operations such as addition, deletion, splitting, and merging across multiple scenarios. As a result, these methods provide bounded support for managing and analyzing evolving, complex software systems [7]. Although the survey provides a thorough outline of PN theory and its extended formalisms, it mainly considers conceptual developments rather than their systematic application to software process evolution. The work does not address how PNs can be structured into modular basic blocks to support process tailoring and evolution. Additionally, there is finite discussion on applying matrix-based or structural analysis techniques to ensure correctness and consistency when PN models are modified or composed in evolving software workflows [8]. Matrix-based PN analysis effectively estimates the structural and behavioral properties, but existing methods consider isolated models rather than evolving software processes. They do not provide systematic paths to assess the correctness and consistency of process tailoring operations: addition, deletion, splitting, and merging. Consequently, there is limited support for using matrix-based techniques to create modular, reusable, and evolving software workflows [13].

3. Software Process Model

Software Evolution is the continuous improvement of a software system after its release, including adding features, fixing defects, and redesigning to meet changing requirements. It makes sure long-lived systems remain usable, compatible, and aligned with user needs. Lehman's Laws illustrate common patterns in how software evolves over time. Software Process Tailoring is the modification of a standard development process to fit the specific requirements of a project or organization. Teams adapt workflows, practices, documentation, and phases to match project size, complexity, team skills, and organizational constraints. Contrasting software evolution, process tailoring considers how the software is developed, not on the software itself. In [15], the software process model is defined by applying five fundamental elements: activity, resource, product, role, and project. An activity represents a unit of work performed by an individual to produce a software product. Each activity contains attributes that capture information relevant to software development. Additionally, activities are governed by constraints that specify their relationships with other objects. A precondition specifies the requirements that must be satisfied before an activity can begin, whereas a postcondition specifies the requirements that must be met upon its completion. A resource is any entity, human or non-human, used to carry out activities. Resources may include personnel, tools, or machines. The concept of resource availability refers to the amount of time that a resource is capable of performing scheduled activities. The output produced by an activity is called a product, which provides a quantitative measure of progress within the development process. Employees may assume multiple roles within the organizational or structural context of a software project. The project object encompasses all activities, resources, and products involved in software development. Accordingly, the flow relationships among activities, their constraints, and the resources required to execute them are illustrated in Figure 1.

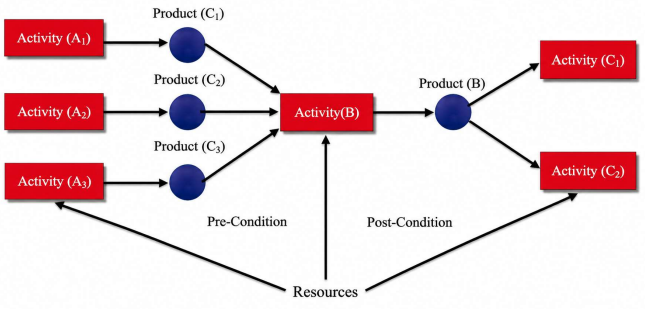

Fig. 1 The flow relationships among activities, constraints, and resources in a software evolution process, illustrating how tasks are coordinated, constrained, and supported throughout the evolution lifecycle

4. Software Evolution Process

Definition 1: The Software evolution process P is a four-tuple, $P = C, A, F, M_0)$, where C is the finite set of conditions, i.e., $C = \{c_1, c_2, \ldots c_n\}$. A is the finite set of activities,

i.e., $A = \{a_1, a_2, \ldots a_m\}$. F is the finite set of flow relations or arcs or forms, i.e., specification document, source code, test report, etc. The flow relationship between activities is determined by the connection of conditions. M_0 is the set of case sets corresponding to the activity.

The software process models are to be modeled by a PN structure. There are four important basic software process blocks, which are all described by PN. These basic blocks are: Sequence or Series, Concurrency or Parallel, Selection or Choice, and Iteration or Loop. This formal representation provides a structured way to model the progression, dependencies, and verification points within the software evolution process.

4.1. Basic Blocks of the Software Evolution Process

Software process models can be effectively represented using PN structures, which provide a formal and graphical way to capture process dynamics. Four fundamental building blocks serve as the basis for modeling software evolution processes with PNs: Sequence or Series Block, Concurrency or Parallel Block, Selection or Choice Block, and Iteration or Loop Block.

Sequence Block: This block models activities executed in a strict linear order, where one activity starts only after the previous one finishes. Also, it represents a series of workflows where tasks are dependent sequentially.

Concurrency Block: This block models activities executed in parallel order execution paths, allowing multiple activities to proceed independently or simultaneously. Also, it is represented by a transition that splits tokens into multiple places (fork), enabling parallel branches, and by a synchronization point (join) where these branches later merge in PN. This shows the essence of multitasking and concurrent processes in software evolution.

Selection Block: This block models capture decision-making points within the process, where one of various alternative paths is selected based on conditions or events. Also, it is represented by a place connected to multiple transitions, where only one transition fires depending on the enabled conditions in PN. Thus, modeling conditional branching.

Iteration Block: This block models a loop or repeated execution of activities until a particular condition is satisfied. In PNs, iteration is depicted by loops where tokens circulate through a set of places and transitions repeatedly until exit conditions enable progression. Also, it reflects repeated testing, reviews, or refinement phases in software evolution.
Hence, these blocks provide a modular framework for illustrating complex software evolution processes. Applying PNs to represent these basic blocks facilitates precise modeling of control flow, synchronization, and decision-making, thereby enabling formal analysis and verification of software process behaviors.

4.2. Petri Net Analysis of Basic Blocks

PNs are a formal and graphical modeling tool widely applied for representing the dynamic behaviors in systems, including software evolution processes. They consist of places (or conditions), transitions (or activities), and arcs or flows that illustrate the state and flow of activities. Tokens within the places indicate the current state of the system and move according to the firing of transitions, enabling analysis of control flow, synchronization, and process dynamics.

Definition 2**:** A Petri net N is a quadruple $\mathrm{N} = (\mathrm{C}, \mathrm{A}, \mathrm{F}, \mathrm{I})$, where C is the finite set of conditions, $\mathrm{C} = \{\mathrm{c}_1, \mathrm{c}_2, \ldots \mathrm{c}_\mathrm{n}\}$. A is the finite set of activities, $\mathrm{A} = \{\mathrm{a}_1, \mathrm{a}_2, \ldots \mathrm{a}_\mathrm{m}\}$. $\mathrm{F} \subseteq (\mathrm{C} \times \mathrm{A}) \cup (\mathrm{A} \times \mathrm{C})$ is the flow relation between condition and activity. $\mathrm{I} \subseteq \mathrm{C}$ is the set of initial conditions.

Mapping Software Process to Petri Net Components**:** In PN terminology, there is a direct and formal correspondence between elements of a software process and components of a PN. The places (or Conditions (C)) in a software process, representing states that must be satisfied, are modeled as places (P) in the PN. The transitions (or activities (A)), which are actions that change the state of the process, are represented as transitions (T). The arc or flow relation (F) specifies the connections between places and transitions, showing which conditions are required before an activity can execute and which conditions are produced afterward. The marking (M) represents the current state of the software process by indicating the distribution of tokens across places, while the initial marking (M_0) defines the starting positioning from which the process begins. Thus, the mapping is illustrated in Table 2.

Table 2. Summarizes this correspondence, providing a concise reference for how software process elements map onto a Petri net

Software Process	Petri net	Description
Condition (C)	Place (P)	Represents a state or prerequisite of the process
Activity (A)	Transition (T)	Represents an action or task that changes the state.
Flow relation (F)	Arcs	Defines input/output relationships between places and transitions.
Process state	Marking (M)	Distribution of tokens over places.
Initial state (I)	Initial marking (M_0)	Starting configuration of the process.

Petri Net Analysis of Sequence Block: In the software evolution process, the well-planned execution of activities often follows a strict sequential pattern, where each activity starts only after the preceding one has completed. This linear ordering of tasks is captured by the method of a Sequence Block, which is essential for modeling deterministic workflows.

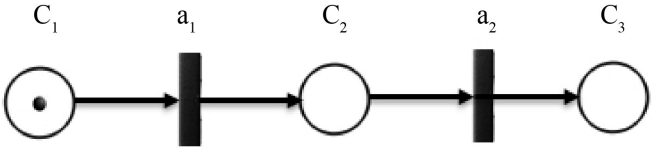

Fig. 2 Representation of sequential execution in a PN, showing how activities occur one after another based on condition dependencies.

Figure 2 describes the Sequence Block using a PN analysis. A PN is a graph consisting of *places* (or conditions or states) and *transitions* (activities), connected by directed arcs or flow representing the flow relation. In Figure 2, the c_1, c_2, and c_3 denote conditions or places in the shape of circles, while the $a_1$and $a_2$represent activities or transitions in the shape of rectangles or bars.

The flow of execution is modeled by the movement of tokens through these conditions c_1, c_2, c_3 and activities a_1, a_2

. Initially, a token occupies a place c_1, indicating that the first condition is satisfied and $a_1$can be enabled. Once $a_1$fires (executes), the token moves to c_2, enabling the next activity a_2. When $a_2$fires, the token advances to c_3, representing the completion of the sequence.

From Figure 2, we get $\mathrm{C} = \{\mathrm{c}_1, \mathrm{c}_2, \mathrm{c}_3\}$, $\mathrm{A} = \{\mathrm{a}_1, \mathrm{a}_2\}$, $F = \{(\mathrm{c}_1, \mathrm{a}_1), (\mathrm{a}_1, \mathrm{c}_2), (\mathrm{c}_2, \mathrm{a}_2), (\mathrm{a}_2, \mathrm{c}_3)\}$, $\mathrm{I} = \big(\mathrm{I}(\mathrm{c}_1), \mathrm{I}(\mathrm{c}_2), \mathrm{I}(\mathrm{c}_3)\big) = (1,0,0)$.

This PN representation captures the inherent sequential dependency between activities, making up that a_2 cannot begin before $a_1$completes, thereby preserving the order of execution fundamental to many software evolution processes. Moreover, this model enables formal analysis of system properties such as reachability, deadlock-freeness, and liveness, which are critical for confirming the correctness and robustness of the evolving software.

Petri Net Analysis of Concurrency Block**:** In the software evolution process, a concurrency block represents the execution of multiple activities in parallel. It models circumstances where two or more activities can start at the same time and proceed independently before eventually synchronizing again.

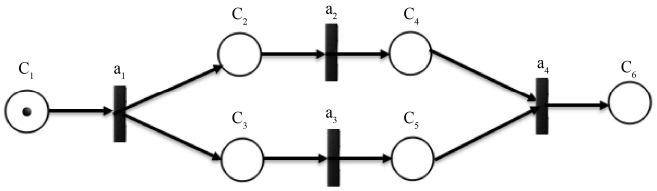

Fig. 3 The representation of concurrency or parallel execution in a PN

Figure 3 describes the structure of concurrency or parallel execution. A token (i.e., c_1) from the initial place is split into two concurrent paths by a transition (representing the start of parallel activities). Each branch contains its own sequence of conditions (or places) and activities (or transitions), modeling independent task flows. Eventually, both branches converge at a synchronization transition, ensuring that all parallel tasks are completed before the process continues. From Figure 3, we get,

$\mathrm{C} = \{\mathrm{c}_1, \mathrm{c}_2, \mathrm{c}_3, \mathrm{c}_4, \mathrm{c}_5, \mathrm{c}_6\}$, $A = \{a_1, a_2, a_3, a_4\}$, $\mathrm{F} = \{(\mathrm{c}_1, \mathrm{a}_1), (\mathrm{a}_1, \mathrm{c}_2), (\mathrm{a}_1, \mathrm{c}_3), (\mathrm{a}_2, \mathrm{c}_3), (\mathrm{c}_2, \mathrm{a}_2), (\mathrm{c}_3, \mathrm{a}_3), (\mathrm{a}_2, \mathrm{c}_4), (\mathrm{a}_3, \mathrm{c}_5), (\mathrm{c}_4, \mathrm{a}_4), (\mathrm{c}_5, \mathrm{a}_4), (\mathrm{a}_4, \mathrm{c}_6)\}$

$\mathrm{I} = \big(I(c_1), I(c_2), I(c_3), I(c_4), I(c_5), I(c_6)\big) = (1,0,0,0,0,0)$.

This PN-based representation provides a clear and formal way to analyze concurrency, detect possible conflicts, and verify correct process behavior in software evolution process workflows.

Petri Net Analysis of Selection Block**:** In the software evolution process, a Selection Block represents the controlled execution of activities where only one among several possible paths is chosen. Figure 3 illustrates the structure of decision-making within a workflow, allowing the process to branch based on conditions or criteria.

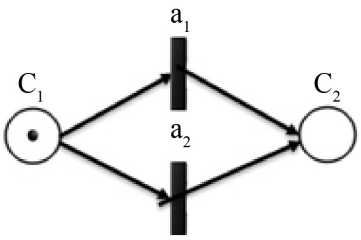

Fig. 4 The representation of selection or choice execution in a PN

The results in PN representation, an initial place distributes a token to one of two transitions (i.e., $\{a_1, a_2\}$) each leading to a distinct activity path. Only the enabled transition fires, making up that exactly one branch is executed. This model helps clearly show the choice, alternative behaviors, and conditional execution within software evolution processes.

From Figure 4, the sets are defined as $\mathrm{C} = \{\mathrm{c}_1, \mathrm{c}_2\}$ is the set of conditions or places, $\mathrm{A} = \{\mathrm{a}_1, \mathrm{a}_2\}$ is the set of activities or transitions $\mathrm{F} = \{(\mathrm{c}_1, \mathrm{a}_1), (\mathrm{c}_1, \mathrm{a}_2), (\mathrm{a}_1, \mathrm{c}_2), (\mathrm{a}_2, \mathrm{c}_2)\}$, $\mathrm{I} = \big(\mathrm{I}(\mathrm{c}_1), \mathrm{I}(\mathrm{c}_2)\big) = (1,0)$ is the initial marking. This model selectively executes by allowing, effectively, only one transition to occur, guiding the process along a single select path.

Petri Net Analysis of an Iteration Block**:** In the software evolution process, an Iteration Block represents a loop structure. Hence, it is also called a loop block, which is the repeated execution of activities until a certain condition is met. Figure 5 illustrates that this structure captures cyclic behavior within a workflow, where a sequence of actions may be executed multiple times before the process proceeds to the next stage.

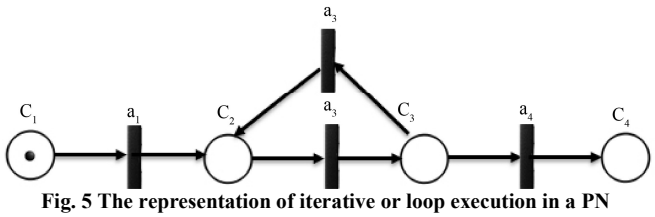

Fig. 5 The representation of iterative or loop execution in a PN

In PN representation, tokens circulate through a loop composed of conditions (or places) and activities (or transitions), enabling repeated firing of particular transitions. This loop continues as long as conditions allow the return path to be enabled. When the iteration condition is satisfied, the flow exits the loop, moving the process toward completion.

Definition 3**:** An incidence matrix $\mathrm{IM}_{|\mathrm{C}|\times|\mathrm{A}|}$ of a Petri net N = (C, A, F, I) is given by

$$m_{ij} = \begin{cases} -1 & (c_j, a_i) \in F, \\ 1 & (a_i, c_j) \in F, \\ 0 & \text{Otherwise} \end{cases}$$

where cell m_{ij} of matrix $IM_{|P|\times|T|}$ refers to activity a_i and condition c_j.

Definition 4**:** A condition invariant C (c-invariant) of a Petri net N = (C, A, F, I) is an integer vector such that $IM \times I^T = 0$.

Definition 5**:** A Petri net N = (C, A, F, I) is covered by condition invariants if every condition $c \in C$ belongs to at least one c-invariant.

Definition 6**:** A Petri net N = (C, A, F, I) is said to be bounded if there is no marking (condition) I such that any condition $c \in C$ contains more than a finite number of tokens. A Petri net N bounded for any finite initial condition I is said to be structurally bounded.

Theorem 1

A Petri net N = (C, A, F, I) is structurally bounded if it is covered by a c-invariant.

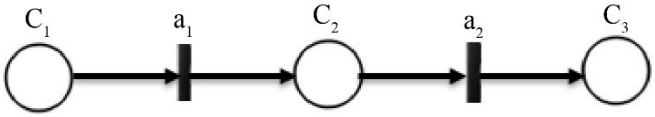

Fig. 6 The sequence block is structurally bounded.

From Figure 6, we prove c-invariant $IM \times I^T = 0$,

Incidence Matrix = $m_{ij} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$,

$$\Rightarrow m_{ij} \times I^T = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times [c_1\ c_2\ c_3]$$

$$\Rightarrow -c_1 + c_2 = 0$$

$$\Rightarrow c_1 = c_2$$

$$\Rightarrow -c_2 + c_3 = 0$$

$$\Rightarrow c_2 = c_3$$

$c_1 = c_2 = c_3 = 1$, for structurally bounded.

Hence, $\begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times [1\ 1\ 1] = [0\ 0]$ is proved.

4.3. Analysis Approaches of Petri Net for Software Evolution Process

In the Software Evolution Process, PNs serve as an effective modeling tool to represent the dynamic behavior and structural dependencies of evolving software systems. As shown in Table 3, three principal analysis approaches are commonly employed to evaluate PN models: Behavioural, Structural, and reduction or refinement approaches. The behavioural approach inspects the execution dynamics of the software system by generating a reachability tree, thereby enabling the analysis of effects such as liveness, boundedness, and deadlock-freeness. The structural approach considers the static organization of the net applying matrix-based techniques, which help identify invariants, conservation laws, and structural consistency without requiring full state-space exploration.

The refinement approach simplifies complex nets by the systematic net-reduction rules while maintaining essential behavioral properties, thus improving scalability and facilitating modular analysis. Together, these approaches enable comprehensive evaluation of software evolution models, ensuring correctness, reliability, and maintainability across the SDLC.

Table 3. The overview of PN approaches for software process modeling highlights their key characteristics and applications.

Sl. No.	Approach	Description
1	Behavioural	It is based on a Tree Structure.
2	Structural	It is based on a Matrix Structure.
3	Reduction or Refinement	It is based on the Net Simplification approach.

In a structural approach, the State equation is an equation that is based on control theory. We consider the elementary firing or control vector denoted by U_k as a $(t \times 1)$ column vector containing exactly one nonzero entry 1 in the position corresponding to the activity fired at the k^{th} firing. From the definition of firing, it is easily seen that the condition m_{k+1} resulting from another condition m_k by the k^{th} firing U_k can be given in terms of the following matrix state equation for a discrete-time system:

$$m_{k+1} = m_k + A\,U_k, \text{ where } k = 0,1,2 \ldots. \quad (1)$$

Where A is the transpose of the activity-to-condition incidence matrix, and its entry $m_{ij} = -1, 1\ or\ 0$ if activity i has an outgoing arc to condition j, an incoming arc from condition j, or no arc between them. respectively. In other words, the i^{th} row of A denotes the token changes in the C conditions when activity i fires once.

Since the number of tokens is a negative integer, the role of the vector U_k is to make the State equation a $(n \times 1)$ matrix of non-negative integers, i.e.,

$$m_k + A\,U_k \geq 0\ \forall\ k = 0,11,2, \ldots \quad (2)$$

The equation (2) can be used to test whether a given firing vector is legal or not, with respect to some marking m_k . The vector U_k controls the validity of the next marking state with respect to the present state. It is reminiscent of the control vector in the state equation in control theory. Hence, the name control vector.

Reachability: Necessary Condition: In reachability, the final marking M_f is reachable from a given initial marking M_0. Suppose a particular firing given by the firing vector U_0 brings the systems from M_0 to M_1, U_1 brings M_1 to M_2,… U_{f-1} brings M_{f-1} to M_f.

Mathematically, these are achieved by repeatedly using the state equation:

$$M_1 = M_0 + A\,U_0$$
$$M_2 = M_1 + A\,U_1$$
$$\dots$$
$$M_f = M_{f-1} + A\,U_f \quad (3)$$

Hence, the final marking M_f is reachable from the initial marking M_0 through a firing sequence $\{U_0, U_1, \dots, U_{f-1}\}$.

$$M_f = M_{f-1} + A\,(U_0 + U_1 + U_2 + \cdots + U_{f-1}$$
$$M_f = M_{f-1} + A\sum_{k=0}^{f-1} U_k$$
$$M_f = M_0 + Ax\,, where\ x = A\sum_{k=0}^{f-1} U_k$$
$$M_f - M_0 = Ax$$
$$\triangle M = Ax\ , where\ \ \triangle M = M_f - M_0$$
$$\triangle M = Ax \quad (4)$$

$\triangle M$ is a $(n \times 1)$ column vector, and x is the sum of all the individual $(m \times 1)$ firing vectors. Control vectors. Also, x is called the firing count vector, which is a non-negative integer where j^{th} entry of x denotes the number of times transition j would fire in a firing sequence leading from M_0 to M_f. Hence, the state equation in general and equation (4), which relates PN state or marking, which is a more dynamic property, with the incidence matrix, and also the representation of the Static PN Structure.

5. Analysis of Structural Blocks in the Software Evolution Process

The tailoring of a software evolution process requires a systematic understanding of the structural patterns that define the flow of activities. These structures, commonly referred to as basic blocks, serve as primary building units for designing, analyzing, and optimizing process models. Representation of these blocks not only upgrades process transparency but also supports consistency, repeatability, and adaptability in evolving software systems. Four principal basic blocks are widely recognized in the software tailoring process: Sequence or Series Block, Concurrency or Parallel Block, Selection or Choice Block, and Iteration or Looping Block. Each block characterizes a specific mode of process execution and contributes uniquely to shaping an effective and flexible evolution process.

5.1. Sequence or Series Block

The execution of activities is sequentially shown in Figure 2. According to the PN model, the places defined as conditions, the activities defined as transitions, and all definitions can be considered.

We can apply the incidence matrix, state equation, and reachability for the Sequence or Series block.

- Incidence matrix: $m_{ij} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$
- State Equation: The sequence block shown in Figure 2 to control the vector in the state equation in control theory is given by $m_{k+1} = m_k + A\,U_k$.

Here, $m_0 = (10\ 0)^T = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, the a_1 is firing

i.e., $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$,

$$m_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},$$

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Longrightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \geq 0$$

Similarly, the a_1 is firing

i.e., $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$,

$$m_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

$$m_1 + A\,U_1 \geq 0$$

$$\Longrightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

- Reachability Necessary Condition: We apply the reachability for the sequence block and find its necessary condition

i.e.,

$$M_f = M_0 + Ax\,, \qquad where\ x = A\sum_{k=0}^{f-1} U_k$$
$$\triangle M = Ax\ , where\ \ \triangle M = M_f - M_0$$
$$\triangle M = Ax$$

Here, f=2, $\triangle M = M_2 - M_0 = Ax$

$$\Rightarrow \triangle M = M_2 - M_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

and

$$Ax = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}.$$

Hence $\triangle M = Ax$ is proved.

5.2. Concurrency or Parallel Block

The concurrency block represents the situation where two or more activities can execute in parallel, without depending on each other's completion, as shown in Figure 3. In the PN formalism, concurrency naturally arises when multiple transitions are enabled at the same time, and their firing does not conflict over shared input places. In this structure, Places represent the *conditions* required for parallel activities to begin. Transitions represent the *independent activities* that may proceed concurrently. Forking (split) occurs when a transition produces tokens for multiple output places, enabling multiple downstream transitions to occur simultaneously. Joining (synchronization) occurs when a transition requires tokens from multiple places before firing, thus ensuring that all parallel activities have completed.

The key property of this block is that the PN allows multiple transitions to fire at the same time, as long as their enabling conditions are satisfied. This freedom to fire independent transitions at the same step captures the inherent parallelism of the process being modeled. Contrasting sequential blocks, the concurrency block enables the system to evolve independently along different branches, each representing parallel software evolution tasks (e.g., independent module updates or maintenance activities).

We can apply the incidence matrix, state equation, and reachability for the Concurrency or Parallel Block.

- Incidence matrix: $m_{ij} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- State Equation: The concurrency block shown in Figure 3 to control the vector in the state equation in control theory is given by $m_{k+1} = m_k + A\,U_k$.

Here, $m_0 = (10\ 0\ 0\ 0\ 0)^T = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

then a_1 is firing i.e., $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

$$m_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \geq 0$$

After a_1 is firing, a forking (split) occurred. Now we get a split of c_2 and c_3 , thus, we calculate the state equation for m_2 and m_3 is as follows:

Here, $m_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

then a_2 is firing i.e., $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$,

$$m_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \geq 0$$

Here, $m_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

then a_3 is firing i.e., $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$,

$$m_3 = \begin{bmatrix}0\\1\\1\\0\\0\\0\end{bmatrix} + \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\\1\\0\end{bmatrix},$$

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix}0\\1\\1\\0\\0\\0\end{bmatrix} + \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}0\\0\\1\\0\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\\1\\0\end{bmatrix} \geq 0$$

After a_2 and a_3 is firing, there is joining (synchronization) that occurs. Now we are joining an activity a_4, thus, we calculate the state equation for m_4 and m_5 is as follows:

Here, $m_2 = \begin{bmatrix}0\\0\\1\\1\\0\\0\end{bmatrix}$,

then a_4 is firing i.e., $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$,

$$m_4 = \begin{bmatrix}0\\0\\1\\1\\0\\0\end{bmatrix} + \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\\-1\\1\end{bmatrix},$$

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix}0\\0\\1\\1\\0\\0\end{bmatrix} + \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\\-1\\1\end{bmatrix} \geq 0$$

Here, $m_3 = \begin{bmatrix}0\\1\\0\\0\\1\\0\end{bmatrix}$,

then a_4 is firing

i.e., $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$,

$$m_4 = \begin{bmatrix}0\\1\\0\\0\\1\\0\end{bmatrix} + \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\-1\\0\\1\end{bmatrix},$$

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix}0\\0\\1\\1\\0\\0\end{bmatrix} + \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\\-1\\1\end{bmatrix} \geq 0$$

At the synchronization (joining) transition, the state equation m_4 and m_5 must satisfy the non-negativity constraint $m_k + Au_k \geq 0$. When one of the parallel branches has not completed, the required tokens are unavailable, producing a negative marking in the corresponding place. Hence, the inequality is violated, indicating that the join transition is not enabled and cannot fire until all parallel activities have completed.

- Reachability Necessary Condition: We apply the reachability for the concurrency block and find its necessary condition, i.e.,

$$M_f = M_0 + Ax, \qquad where\ x = A\sum_{k=0}^{f-1} U_k$$

$\triangle M = Ax$, $where\ \ \triangle M = M_f - M_0$

$\triangle M = Ax$

Here, f=4, $\triangle M = M_4 - M_0 = Ax$

$$\Rightarrow \triangle M = M_4 - M_0 = \begin{bmatrix}0\\0\\1\\0\\-1\\1\end{bmatrix} - \begin{bmatrix}1\\0\\0\\0\\0\\0\end{bmatrix} = \begin{bmatrix}-1\\0\\1\\0\\-1\\1\end{bmatrix}$$

and

$$Ax = \begin{bmatrix}-1&0&0&0\\1&-1&0&0\\1&0&-1&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix} \times \begin{bmatrix}1\\1\\1\\1\end{bmatrix} = \begin{bmatrix}1\\0\\1\\0\\0\\1\end{bmatrix}.$$

Here, f=5, $\triangle M = M_5 - M_0 = Ax$

$$\Rightarrow \triangle M = M_5 - M_0 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ 0 \\ -1 \\ 0 \\ 1 \end{bmatrix}$$

And

$$Ax = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

For the concurrency block, the reachability condition requires that $\Delta M = Ax$, where $x = A\sum_{k=0}^{f-1} U_k$ includes the firing of all transitions in both parallel branches. If one of the concurrent transitions does not fire, the resulting firing count vector x'does not satisfy the reachability equation. In such a case, the marking produced by Ax'cannot generate the token configuration required for the synchronization transition, since the join requires tokens from all parallel branches. Consequently, $m_k + Au_k \geq$/0at the join, meaning the final marking $M_f$is not reachable, and the concurrency block is not satisfied. Hence $\triangle M = Ax$ is proved.

This makes Petri nets an effective model for describing parallelism, synchronization, and distributed behavior within software evolution processes.

5.3. Selection or Choice Block

The Selection or Choice Block models a point where the system must choose one among several possible actions, as shown in Figure 4. In PNs, a single input place connects to multiple alternative transitions. Only one transition can fire, representing the chosen activity, while the others remain inactive due to token consumption. This model has exclusive branching within the workflow.

We can apply the incidence matrix, state equation, and reachability for the Selection or Choice Block.

- Incidence matrix: $m_{ij} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$
- State Equation: The selection block shown in Figure 4 to control the vector in the state equation in control theory is given by $m_{k+1} = m_k + A\,U_k$.

Here, $m_0 = (10)^T = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$,

then a_1 is firing

i.e., $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $m_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$,

$m_k + A\,U_k \geq 0$

i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \geq 0$$

Similarly, the a_2 is firing

i.e., $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$,

$$m_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

$$m_1 + A\,U_1 \geq 0$$

$$\Rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

- Reachability Necessary Condition: We apply the reachability for the Selection or Choice Block and find its necessary condition

i.e.,

$$\mathrm{M_f} = \mathrm{M_0} + \mathrm{Ax}, \qquad \text{where } \mathrm{x} = \mathrm{A}\sum_{\mathrm{k=0}}^{\mathrm{f-1}} \mathrm{U_k}$$

$\triangle \mathrm{M} = \mathrm{Ax}$, where $\triangle \mathrm{M} = \mathrm{M_f} - \mathrm{M_0}$ $\triangle \mathrm{M} = \mathrm{Ax}$

Here, f=1, $\triangle \mathrm{M} = \mathrm{M_1} - \mathrm{M_0} = \mathrm{Ax}$

$$\Rightarrow \triangle \mathrm{M} = \mathrm{M_1} - \mathrm{M_0} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

and

$$\mathrm{Ax} = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \end{bmatrix}.$$

In the Selection or Choice Block, only one transition may fire at a time. The reachability equation requires that $\Delta M = Ax$. However, the marking difference $\Delta M = [-1\ 1]^T$cannot be generated by the incidence matrix when both alternative transitions are included in the firing vector.

Since a choice block permits only one successor transition to fire, the firing vector $x = [1\ 1]^T$is invalid, and its product $Ax = [-2\ 2]^T$does not match the required marking change. Thus, the necessary reachability condition is not satisfied, indicating that the computed marking is not reachable in a valid choice configuration. Hence $\triangle M = Ax$ is proved.

5.4. Iteration or Loop Block

The Iteration or Loop Block represents the repeated execution of a set of activities shown in Figure 5. In a Petri net model, this repetition is captured structurally by defining places that represent the loop conditions and transitions that represent the activities executed during each iteration. A token placed in the loop-condition place enables the transitions inside the loop, allowing them to fire repeatedly as long as the loop condition remains true.

Each firing of a transition corresponds to one cycle of the loop. When the specified number of iterations is completed, or when the exit condition becomes satisfied, the loop-exit transition fires, removing the token from the loop and allowing control to move to the next part of the process. This models the classical looping behavior within the Petri net framework.

We can apply the incidence matrix, state equation, and reachability for the Iteration or Loop Block.

- Incidence matrix: $m_{ij} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- State Equation: The iteration block shown in Figure 5 to control the vector in the state equation in control theory is given by $m_{k+1} = m_k + A\,U_k$.

Here, $m_0 = (10\ 0\ 0)^T = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, then a_1 is firing

i.e., $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

$$m_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

$m_k + A\,U_k \geq 0$
i.e., $m_0 + A\,U_0 \geq 0$

$$\Rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \geq 0$$

The a_2 is firing i.e., $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$,

$$m_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$m_1 + A\,U_1 \geq 0$

$$\Rightarrow \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \geq 0$$

The a_3 is firing
i.e., $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$,

$$m_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$m_2 + A\,U_2 \geq 0$

$$\Rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \geq 0$$

The a_4 is firing
i.e., $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$,

$$m_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$m_3 + A\,U_3 \geq 0$

$$\Rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \geq 0$$

- Reachability: Necessary Condition: We apply the reachability for the iteration or Loop Block and find its necessary condition, i.e.,

$$\mathrm{M_f} = \mathrm{M_0} + \mathrm{Ax}, \qquad \text{where } \mathrm{x} = \mathrm{A}\sum_{\mathrm{k=0}}^{\mathrm{f-1}} \mathrm{U_k}$$

$\triangle \mathrm{M} = \mathrm{Ax}$, where $\triangle \mathrm{M} = \mathrm{M_f} - \mathrm{M_0}$ $\triangle \mathrm{M} = \mathrm{Ax}$
Here, f=3,
$\triangle \mathrm{M} = \mathrm{M_3} - \mathrm{M_0} = \mathrm{Ax}$

$$\Rightarrow \Delta \mathrm{M} = \mathrm{M_3} - \mathrm{M_0} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

and

$$Ax = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}$$

The reachability condition requires that the change in marking produced by the incidence matrix exactly matches the change between the initial and final markings. However, as demonstrated in the figure, the computed vector Axdoes not equal ΔM. Because the system cannot generate the required marking change through any combination of transition firings shown, the reachability condition is not satisfied, and the final marking is therefore not reachable under the given loop block. Hence $\triangle M = Ax$ is proved.

5.5. Example of Dynamic Web Service Composition Using Basic Blocks of Software Evolution Process

This software evolution model is derived from a PN-based dynamic service composition framework. Activities (a) represent executable steps in the evolution process, while conditions (C) denote decision points that control the execution flow. The evolution process starts with change

information acquisition (a_1), during which change requests and operational feedback are collected. The acquired information is then processed and analyzed (a_2) and forwarded to evolution decision and control (a_3), where an appropriate evolution strategy is determined. The condition node C_4 evaluates the nature of the change. According to this evaluation, the process branches into corrective evolution (a_4) for fault-related modifications or adaptive evolution (a_5) for functionality enhancements or environmental adaptations. Upon execution of the selected evolution activity, evolution effectiveness is evaluated (a_6). If the expected objectives are not met, the process enters a repeat evolution cycle (a_7) until satisfactory results are achieved in Figure 7 and Table 4. The model enables formal verification of key properties such as reachability, safety, and absence of deadlock, thereby ensuring the correctness and reliability of the software evolution process.

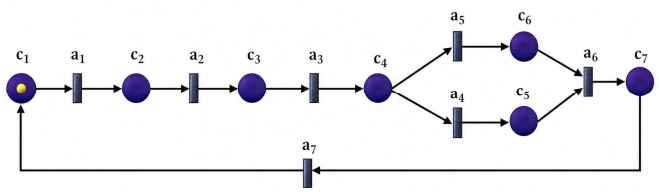

Fig. 7 The PN-based online order business process constructed from basic blocks to support software evolution.

Table 4. The symbols used in the software evolution model, where activities (a) represent evolution actions and conditions (C) define decision points that control the execution and branching of the process flow

Symbol	Operation	Comment
a_1	change information acquisition	basic activity (input)
a_2	change information processing	service invoking
a_3	evolution decision and control	service invoking
c_4	change type condition	branch condition
a_4	corrective evolution execution	service invoking
a_5	adaptive evolution execution	service invoking
a_6	evolution effectiveness evaluation	service invoking

We can apply the incidence matrix, state equation, and reachability for the dynamic web service composition.

- Incidence matrix:

$$m_{ij} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

- State Equation: The dynamic web service composition shown in Figure 7 to control the vector in the state equation in control theory is given by

$m_{k+1} = m_k + A\,U_k$.

From Figure 3, the initial marking

$$m_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

control (firing) vector

$$U_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and

$$A = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

we find,

$$m_1 = m_0 + AU_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Similarly, fire a_2 from marking m_1 we get

$$m_2 = m_1 + AU_1 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Hence, the steps and the final result are summarized in Table 5.

Table 5. Stepwise marking evolution and final state in dynamic web service composition reachability

Step	Fired Activities	Marking m_k
0	—	$[1, 0, 0, 0, 0, 0, 0]^T$
1	a_1	$[0, 1, 0, 0, 0, 0, 0]^T$
2	a_2	$[0, 0, 1, 0, 0, 0, 0]^T$

3	a_3	$[0, 0, 0, 1, 0, 0, 0]^T$
4	a_4	$[0, 0, 0, 0, 1, 0, 0]^T$
4	a_5	$[0, 0, 0, 0, 0, 1, 0]^T$
5	a_6	$[0, 0, 0, 0, 0, 0, 1]^T$
0	—	$[1, 0, 0, 0, 0, 0, 0]^T$
1	a_1	$[0, 1, 0, 0, 0, 0, 0]^T$

6. Operations in Tailoring Evolution Process

In the tailoring software evolution process, there are four operations on tailoring activities. The operations are: *Addition Operation, Deletion Operation, Splitting Operation*, and *Merging Operation*.

Conflict and Confluence Matrices: A PN is a directed graph that is used to describe and analyse the flow of information or resources in a system. It consists of: conditions (C), activities (A), and Arcs or flow relations, which connect conditions to activities and vice versa, and Tokens: Reside in places; their movement represents condition changes.

PN has an adjacency matrix (or a node-to-node incidence matrix) $\mathrm{IM}_{|C|\times|A|} = [m_{ij}]$. The condition for the adjacency matrix is as follows:

$$m_{ij} = \begin{cases} 1 & \text{Vertex i to Vertex j.} \\ 0 & \text{Otherwise.} \end{cases}$$

The adjacency matrix A of a Petri net having n activities and m conditions is a (0, 1)-square matrix of order (n + m), and the formal definition of this matrix is given by

A matrix A= $[a_{ij}] \in \Re^{n\times n}$ is called a (0,1)-square matrix if $a_{ij} \in \{0,1\}\ \forall\ i,j \in \{1,2,\dots,n\}$ i.e., every element of the matrix is either 0 or 1, and the shape of the matrix is $n \times n$. Now, (0, 1)-square matrix of order $(n + m)$ and can be partitioned, given by equation (5).

The matrix Arepresents the *structure of a Petri net* in a compact mathematical form. It is written as a *block matrix* to distinguish between places and transitions and their relationships clearly. The matrix is defined as:

$$A = \begin{bmatrix} 0 & B \\ F & 0 \end{bmatrix} \tag{5}$$

where $n$denotes the number of *places* and $m$denotes the number of *transitions* in the Petri net. We explain the Submatrices:

- *Zero matrices (0)*: The diagonal blocks are zero matrices because places are not directly connected to places, and transitions are not directly connected to transitions in a Petri net.
- *Matrix B(n × m)*: This matrix represents the *input relations* from places to transitions. Each element b_{ij}indicates whether place $i$is an input to transition j. In other words, it shows which conditions must be satisfied before an activity can occur.
- *Matrix F(m × n)*: This matrix represents the *output relations* from transitions to places. Each element f_{ji}indicates whether transition $j$produces a token in place $i$after firing.

Here, B and F are two submatrices and can be defined as $B = [b_{ij}]_{n\times m}$and $F = [f_{ij}]_{m\times n}$. As per two sub-matrices, we need notations, i.e., O(a) is the set of output conditions of activity, I(a) is the set of input conditions of activity, O(c) is the set of output conditions of activity, and I(c) is the set of input conditions of activity c. Then $b_{ij} \neq 0$ iff a condition c_j is in $O(a_i)$ or equivalently, an activity a_i is in $I(c_j)$. Similarly, $f_{ij} \neq 0$ iff a condition c_i is in $I(a_j)$ or equivalently, an activity a_j is in $O(c_i)$.

Definition 7: Activity and Condition conflict matrix

The conflict occurs when multiple activities are enabled by the same place and vice versa. The activity conflict matrix is a $(n \times n)$ symmetric matrix, and the condition conflict matrix is an $(m \times m)$ symmetric matrix. But the product matrix of the transition conflict matrix can be defined in (6), and the product matrix of the place conflict matrix can be defined in (7).

$$F'F = P = [p_{ij}] \tag{6}$$

Where $p_{ij} = |I(a_i) \cap I(a_j)|$ is the number of input conditions that a_i and a_j have in common and $p_{ii} = |I(a_i)|$ is the total number of input conditions of a_i.

$$B'B = Q = [q_{ij}] \tag{7}$$

Where $q_{ij} = |I(c_i) \cap I(c_j)|$ is the number of input conditions that c_i and c_j have in common and $q_{ii} = |I(c_i)|$ is the total number of input conditions of c_i.

Definition 8: Activity and Condition Confluence matrix

The confluence occurs when different sequences of activities lead to the same marking (condition) and vice versa. The activity confluence matrix is a $(n \times n)$ symmetric matrix, and the condition confluence matrix is a $(m \times m)$ symmetric matrix. But the product matrix of the activity confluence matrix can be defined in (8), and the product matrix of the condition conflict matrix can be defined in (9).

$$BB' = R = [r_{ij}] \tag{8}$$

Where $r_{ij} = |O(a_i) \cap O(a_j)|$ is the number of output conditions that a_i and a_j have in common and $r_{ii} = |O(a_i)|$ is the total number of output conditions of a_i.

$$FF' = S = [s_{ij}] \tag{9}$$

Where $s_{ij} = \left|O(c_i) \cap O(c_j)\right|$ is the number of output conditions that c_i and c_j have in common and $s_{ii} = |O(c_i)|$ is the total number of output conditions of c_i.

6.1. Adding Operation

In the software evolution process model, we consider four basic relations, i.e., sequence relation, concurrence relation, selection relation, and iteration relation. In addition, we add the newly active activity with adjacent activity shown in Figure 8(a), which represents one sequence block, and 8(b) represents another sequence block. But the addition operation of two sequence blocks is shown in 8(c).

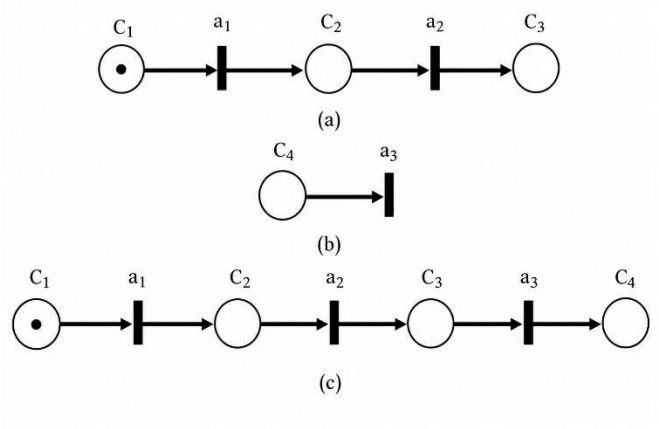

Fig. 8 The addition operation of the sequence block, illustrating how multiple input sequences are element-wise summed to produce a single output representation.

Theorem 2

A Basic Block is a Petri net. But the addition operation of Basic Blocks iff (if and only if) the diagonal elements of (activity-) Conflict and confluence matrices are equal to one. Example of Theorem 2, from Figure 8(c).

The matrix B represents the *input (pre-incidence) relationships* in a PN. It defines how *places (or conditions)* are connected to *transitions (or activities)* and specifies which conditions must be satisfied before an activity can occur. In this matrix: Columns labeled C_1, C_2, C_3, C_4represent *places* and Rows labeled a_1, a_2, a_3represent *transitions*.

$$B = \begin{matrix} \\ a_1 \\ a_2 \\ a_3 \end{matrix} \begin{matrix} C_1 & C_2 & C_3 & C_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

$$B = \begin{bmatrix} 0100 \\ 0010 \\ 0001 \end{bmatrix}, B' = \begin{bmatrix} 000 \\ 100 \\ 010 \\ 001 \end{bmatrix},$$

$$R = BB' = \begin{bmatrix} 100 \\ 010 \\ 001 \end{bmatrix}.$$

Similarly, we find

$$F = \begin{matrix} \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{matrix} \begin{matrix} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix}$$

$$F = \begin{bmatrix} 100 \\ 010 \\ 001 \\ 000 \end{bmatrix}, F' = \begin{bmatrix} 1000 \\ 0100 \\ 0010 \end{bmatrix},$$

$$P = F'F = \begin{bmatrix} 100 \\ 010 \\ 001 \end{bmatrix}$$

Proof: The conflict matrix and the confluence matrix are defined as $P_{ij} = 1$ if the activity from a_i to a_j In conflict, i.e., they share an input condition. But $R_{ij} = 1$ if the activity from a_i to a_j In Confluence, i.e., they share the output condition. So, $P_{ii} = 1$ if the activity from a_i in conflict with at least one input condition. $R_{ii} = 1$ if the activity from t_i in conflict with at least one output condition.

Hence, if all the diagonal elements of P and R are equal, it means every activity has at least one input and one output, which is true for all basic blocks' addition operation.

6.2. Deletion Operation

Theorem 2 gives the structural condition under which adding Basic Blocks preserves Petri-net correctness. By symmetry, the deletion operation must also preserve these structural conditions. In the deletion operation, we delete an activity. Figure 9(a) represents one iteration block, and Figure 9(b) represents the sequence block structure.

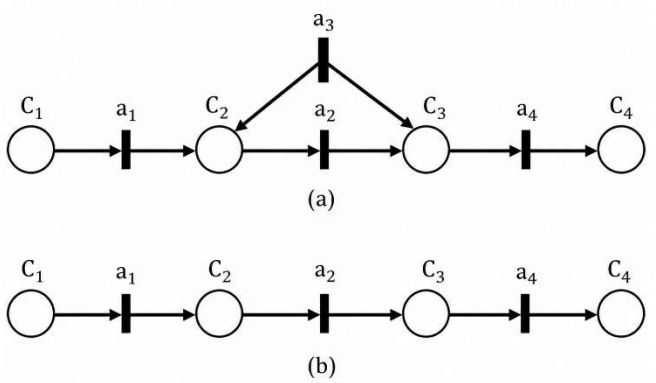

Fig. 9 The deletion operation of basic blocks, where specific blocks are removed from the system, and the remaining blocks are reorganized to ensure continuity of data flow and correct operation.

A Basic Block is a Petri net. But the deletion operation of Basic Blocks iff (if and only if) the diagonal elements of (activity) Conflict and confluence matrices are equal to one. Example of Theorem 2: From Figure 9(b),

$$B = \begin{matrix} \\ a_1 \\ a_2 \\ a_3 \end{matrix} \begin{matrix} C_1 & C_2 & C_3 & C_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

$$B = \begin{bmatrix} 0100 \\ 0010 \\ 0001 \end{bmatrix}, B' = \begin{bmatrix} 000 \\ 100 \\ 010 \\ 001 \end{bmatrix},$$

$$R = BB' = \begin{bmatrix} 100 \\ 010 \\ 001 \end{bmatrix}.$$

Similarly, we find

$$F = \begin{matrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{matrix} \begin{matrix} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{matrix}$$

$$F = \begin{bmatrix} 100 \\ 010 \\ 001 \\ 000 \end{bmatrix}, F' = \begin{bmatrix} 1000 \\ 0100 \\ 0010 \end{bmatrix}, P = F'F = \begin{bmatrix} 100 \\ 010 \\ 001 \end{bmatrix}$$

The Deletion Operation preserves the Basic Block property when the diagonal elements of the reduced Conflict and Confluence matrices remain equal to 1. In the given Petri-net figure 8(a), deleting transition a_3 satisfies this requirement, and therefore the reduced block remains a valid Basic Block according to Theorem 2.

6.3. Merge Operation based on Theorem 2

Theorem 2 gives the structural condition under which adding Basic Blocks preserves Petri-net correctness. By symmetry, the merge operation must also preserve these structural conditions. We take a sequence block from the basic block shown in Figure 10. and another basic block, which is a loop-like choice shown in Figure 11. We merge the two basic blocks shown in Figure 12.

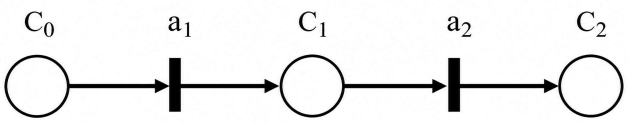

Fig. 10 The sequence block representation of a basic block during the merge operation

From Figure 10, we get,

$$B = \begin{matrix} \\ a_1 \\ a_2 \end{matrix} \begin{matrix} C_0 & C_1 & C_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$B = \begin{bmatrix} 010 \\ 001 \end{bmatrix}, B' = \begin{bmatrix} 00 \\ 10 \\ 01 \end{bmatrix},$$

$$R = BB' = \begin{bmatrix} 010 \\ 001 \end{bmatrix} \times \begin{bmatrix} 00 \\ 10 \\ 01 \end{bmatrix} = \begin{bmatrix} 10 \\ 01 \end{bmatrix}$$

Similarly, we find,

$$F = \begin{matrix} \\ a_1 \\ a_2 \end{matrix} \begin{matrix} C_0 & C_1 & C_2 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{matrix}$$

$$F = \begin{bmatrix} 10 \\ 01 \\ 00 \end{bmatrix}, F = \begin{bmatrix} 100 \\ 010 \end{bmatrix},$$

$$P = FF' = \begin{bmatrix} 10 \\ 01 \\ 00 \end{bmatrix} \times \begin{bmatrix} 100 \\ 010 \end{bmatrix} = \begin{bmatrix} 10 \\ 01 \end{bmatrix}$$

From Figure 11, which represents an iteration or loop-like choice.

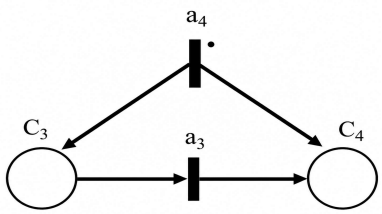

Fig. 11 The Merge operation showing control flow convergence in the iteration (loop-like) choice block

From Figure 11, we find,

$$B = \begin{matrix} \\ a_3 \\ a_4 \end{matrix} \begin{matrix} C_3 & C_4 \\ 1 & 0 \\ 0 & 1 \end{matrix}$$

$$B = \begin{bmatrix} 01 \\ 10 \end{bmatrix}, B' = \begin{bmatrix} 01 \\ 10 \end{bmatrix},$$

$$R = BB' = \begin{bmatrix} 01 \\ 10 \end{bmatrix} \times \begin{bmatrix} 01 \\ 10 \end{bmatrix} = \begin{bmatrix} 10 \\ 01 \end{bmatrix}$$

Similarly, we find,

$$F = \begin{matrix} C_3 \\ C_4 \end{matrix} \begin{matrix} a_3 & a_4 \\ 1 & 0 \\ 0 & 1 \end{matrix}$$

$$F = \begin{bmatrix} 10 \\ 01 \end{bmatrix}, F' = \begin{bmatrix} 10 \\ 01 \end{bmatrix},$$

$$P = FF' = \begin{bmatrix} 10 \\ 01 \end{bmatrix} \times \begin{bmatrix} 10 \\ 01 \end{bmatrix} = \begin{bmatrix} 10 \\ 01 \end{bmatrix}$$

Now, we merge the two basic blocks shown in Figures 10(a) and 10(b) and get the result of the merge operation shown in Figure 12.

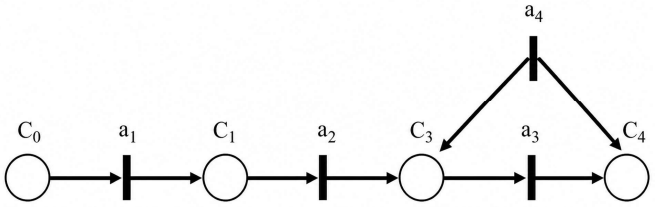

Fig. 12 The Merge operation illustrating the combination of two basic blocks

From Figure 12, we find,

$$B = \begin{matrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{matrix} \begin{matrix} C_0 & C_2 & C_3 & C_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix}$$

$$B = \begin{bmatrix}0100\\0010\\0001\\0010\end{bmatrix},\quad B' = \begin{bmatrix}0100\\1000\\0101\\0010\end{bmatrix},$$

$$R = BB' = \begin{bmatrix}0100\\0010\\0001\\0010\end{bmatrix} \times \begin{bmatrix}0100\\1000\\0101\\0010\end{bmatrix} = \begin{bmatrix}1000\\0100\\0010\\0001\end{bmatrix}$$

Similarly, we find,

$$F = \begin{array}{c} \\ C_0 \\ C_2 \\ C_3 \\ C_4 \end{array} \begin{array}{cccc} a_1 & a_2 & a_3 & a_4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}$$

$$F = \begin{bmatrix}1000\\0100\\0010\\0001\end{bmatrix}, F' = \begin{bmatrix}1000\\0100\\0010\\0001\end{bmatrix},$$

$$P = F'F = \begin{bmatrix}1000\\0100\\0010\\0001\end{bmatrix} \times \begin{bmatrix}1000\\0100\\0010\\0001\end{bmatrix} = \begin{bmatrix}1000\\0100\\0010\\0001\end{bmatrix}$$

Hence, if all the diagonal elements of P and R are equal, it means every activity has at least one input and one output, which is true for all basic blocks' merge operation.

6.4. Split Operation based on Theorem 2

Theorem 2 gives the structural condition under which adding Basic Blocks preserves Petri-net correctness. By symmetry, the split operation must also preserve these structural conditions. We take a sequence block from the basic block shown in Figure 13.

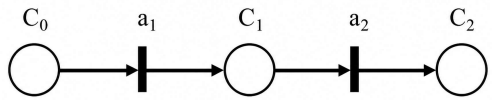

Fig. 13 The Split operation in the sequence block of a basic block

From Figure 13, we find,

$$B = \begin{array}{c} \\ a_1 \\ a_2 \end{array} \begin{array}{ccc} C_0 & C_1 & C_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}$$

$$B = \begin{bmatrix}010\\101\end{bmatrix}, B' = \begin{bmatrix}00\\10\\01\end{bmatrix},$$

$$R = BB' = \begin{bmatrix}010\\101\end{bmatrix} \times \begin{bmatrix}00\\10\\01\end{bmatrix} = \begin{bmatrix}10\\01\end{bmatrix}$$

Similarly, we find,

$$F = \begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \end{array} \begin{array}{cc} a_1 & a_2 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{array}$$

$$F = \begin{bmatrix}10\\01\\00\end{bmatrix}, F' = \begin{bmatrix}100\\010\end{bmatrix},$$

$$P = F'F = \begin{bmatrix}100\\010\end{bmatrix} \times \begin{bmatrix}10\\01\\00\end{bmatrix} = \begin{bmatrix}10\\01\end{bmatrix}$$

We perform a split on transition a_2 so that instead of one outgoing transition, we introduce two alternative branches shown in Figure 13:

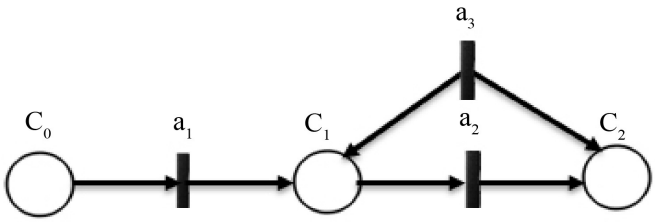

Fig. 14 The Split operation illustrating the sequence block

From Figure 14, we find,

$$B = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \end{array} \begin{array}{ccc} C_0 & C_1 & C_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{array}$$

$$B = \begin{bmatrix}010\\001\\100\end{bmatrix}, B' = \begin{bmatrix}001\\100\\010\end{bmatrix},$$

$$R = BB' = \begin{bmatrix}010\\001\\100\end{bmatrix} \times \begin{bmatrix}001\\100\\010\end{bmatrix} = \begin{bmatrix}100\\010\\001\end{bmatrix}$$

Similarly, we find,

$$F = \begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \end{array} \begin{array}{ccc} a_1 & a_2 & a_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}$$

$$F = \begin{bmatrix}100\\010\\001\end{bmatrix}, F' = \begin{bmatrix}100\\010\\001\end{bmatrix},$$

$$P = F'F = \begin{bmatrix}100\\010\\001\end{bmatrix} \times \begin{bmatrix}100\\010\\001\end{bmatrix} = \begin{bmatrix}100\\010\\001\end{bmatrix}$$

Hence, if all the diagonal elements of P and R are equal, it means every activity has at least one input and one output, which is true for all basic blocks' split operation.

The requirement that the diagonal elements of the conflict and confluence matrices be equal to one verifies that each activity within a software process workflow behaves as a self-consistent and well-defined component of execution. Physically, this condition indirectly implies that an activity neither conflicts with itself nor exhibits ambiguous internal convergence, thereby guaranteeing that it has a unique enabling condition and produces a single, deterministic effect. Such behavior is necessary for preserving the atomicity and predictability of individual tasks, whether they represent coding, testing, review, or other process steps. Furthermore, the condition makes sure that Basic Blocks can be composed through the addition operation without introducing internal inconsistencies, deadlocks, or nondeterministic behavior. Therefore, the resulting PN preserves soundness and reliability when modeling complex software evolution workflows, enabling rigorous analysis, verification, and safe reuse of process components.

Theorem 3

If Basic block is a Petri Net N, then the addition operation of two or more blocks is the (Condition-) conflict and confluence matrices Q and S are diagonal matrices.

Example of Theorem 3: From Figure 8(c),

$$Q = B'B = \begin{bmatrix}100\\010\\001\end{bmatrix}, S = FF' = \begin{bmatrix}100\\010\\001\end{bmatrix}.$$

Poof: The condition conflict matrix Q is $Q_{ij} = 1$ if condition c_i is connected condition c_j by the same output activity, i.e., they are in conflict, are preconditions to the same activity. The condition confluence matrix S is $S_{ij} = 1$ if condition $c_i$and c_j Both receive tokens from the same activity, i.e., they are post-conditions of the same activity. If the Petri net is the addition operation of two Basic Blocks, the activity has exactly one input and one output condition for each activity a. Hence, $Q_{ij} = \lceil I(c_i) \cap I(c_j) \rceil = 0 \; for \; i \neq j$. The activity cannot be in the input set of two different conditions. Since they would both be in O(a). Now, $I(c_1) = \{\emptyset\}, I(c_2) = \{a_1\}$, $I(c_3) = \{a_3\}$, $I(c_4) = \{a_3\}$. $O(c_1) = \{a_1\}$, $O(c_2) = \{a_2\}$, $O(c_3) = \{a_3\}$, $O(c_4) = \{\emptyset\}$. Hence, $D_{ij} = \lceil I(p_{in}) \cap I(p_1) \rceil = \lceil \{\emptyset\} \cap \{t_a\} \rceil = |\emptyset| = 0$. Similarly, $S_{ij} = \lceil O(c_i) \cap O(c_j) \rceil = 0 \; for \; i \neq j$. The activity cannot be in the output set of two different conditions. Since they would both be in I(a). Hence, $S_{ij} = \lceil O(c_1) \cap O(c_2) \rceil = \lceil \{a_1\} \cap \{a_2\} \rceil = |\emptyset| = 0$. Thus, if the addition of both Q and S is a diagonal matrix.

Theorem 4

If both Q and S are diagonal matrices, then either the addition operation of Basic Blocks N or there exists an activity a with either no input /output conditions.

Poof: If two matrices Q and S are diagonal matrices, then $\lceil I(c_i) \cap I(c_j) \rceil = 0 \; for \; i \neq j$ and $\lceil O(c_i) \cap O(c_j) \rceil = 0 \; for \; i \neq j$. Thus, for any activity a, I(a) and O(a) can contain at most one place each. Hence, either N is an addition operation of Basic Block, or for some activity, either $|\mathrm{I(a)}| = 0$ or $|\mathrm{O(a)}| = 0$.

7. Conclusion

In this paper, we have presented a structured approach for customizing the software evolution process. This approach introduces four fundamental tailoring operations: addition, deletion, splitting, and merging based on basic building blocks. Furthermore, when performing splitting or merging operations, consistency between the high-level model and the low-level model is maintained. Petri nets are considered an effective modeling tool for systems that require the identification of basic blocks of the software evolution process.

Further, we have presented multiple matrix representations of Petri nets and discussed their potential uses. These matrix-based methods can support the analysis of different subclasses of Petri nets. Future work will focus on advancing the understanding and practical application of process tailoring. Specifically, research will investigate how organizational context, capability levels, and environmental constraints influence tailoring decisions. Experimental studies will be managed to validate proposed tailoring strategies and evaluate their impact on project performance. These efforts aim to initiate clearer guidance for practitioners and support the development of more flexible and effective process frameworks.

Acknowledgments

Sincere gratitude is extended to Prof. Jibendu Kumar Mantri, Department of Computer Applications, MSCB University, for his invaluable guidance, constructive feedback, and continuous encouragement throughout this research. His expertise in software engineering, particularly in software evolution, process modeling, and analytical techniques, significantly contributed to the development and refinement of this work. Appreciation is also expressed for his insightful suggestions in shaping the conceptual framework, including the application of modeling approaches such as Petri nets and process evolution methodologies, which enhanced the depth and rigor of the study. Gratitude is further extended to the Department of Computer Applications, MSCB University, for providing a supportive academic environment and the necessary resources that facilitated this research on software evolution and its associated modeling techniques.

References

[1] Miryung Kim, Na Meng, and Tianyi Zhang, "Software Evolution," *Handbook of Software Engineering*, pp. 223-284, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[2] Michael W. Godfrey, and Daniel M. German, "On the Evolution of Lehman's Laws," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 613-619, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[3] Andy Zaidman, Martin Pinzger, and Arie van Deursen, "Software Evolution," *Encyclopedia of Software*, 2010. [Google Scholar] [Publisher Link]

[4] Tom Mens, and Serge Demeyer, *Software Evolution*, Springer-Verlag Berlin Heidelberg, 2008. [Publisher Link]

[5] D. Welzel, H.L. Hausen, and W. Schmidt, "Tailoring and Conformance Testing of Software Processes: The Procept Approach," *Proceedings of the Software Engineering Standards Symposium*, pp. 41-49, 1995. [CrossRef] [Google Scholar] [Publisher Link]

[6] Anthony Spiteri Staines, "Representing Petri Net Structures as Directed Graphs," *Proceedings of the 10th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, 2011. [Google Scholar] [Publisher Link]

[7] Edgar Sarmiento, Julio Cesar Sampaio Do Prado Leite, and Eduardo Almentero, "Analysis of Scenarios with Petri-Net Models," *Proceedings of the 29th Brazilian Symposium on Software Engineering*, pp. 90-99, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[8] T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, 1989. [CrossRef] [Google Scholar] [Publisher Link]

[9] Robert H. Martin, and David Raffo, "A Model of the Software Development Process Using both Continuous and Discrete Models," *Software Process Improvement and Practice*, vol. 5, pp. 2-3, pp. 147-159, 2000. [CrossRef] [Google Scholar] [Publisher Link]

[10] Neil A. Ernst, Alexander Borgida, and John Mylopoulos, "Requirements Evolution Drives Software Evolution," *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pp. 16-20, 2011. [CrossRef] [Google Scholar] [Publisher Link]

[11] Shang Zheng, and Hongji Yang, "A Three-Dimensional Approach to Evolving Software," *IEEE 37th Annual Computer Software and Applications Conference Workshops*, pp. 475-480, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[12] Fei Dai, and Tong Li, "Tailoring Software Evolution Process," *Proceedings of the Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pp. 264-269, 2007. [CrossRef] [Google Scholar] [Publisher Link]

[13] J.L. Johnson, and T. Murata, "Structure Matrices for Petri Nets and their Applications," *Journal of the Franklin Institute*, vol. 319, no. 3, pp. 299-309, 1985. [CrossRef] [Google Scholar] [Publisher Link]

[14] Sayed Moshin Reza et al., "Performance Analysis of Machine Learning Approaches in Software Complexity Prediction," *Proceedings of International Conference on Trends in Computational and Cognitive Engineering*, pp. 27-39, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[15] A. Hazeyama, and S. Komiya, "A Process Model for Software Process Management," *Proceedings Fourth International Conference on Software Engineering and Knowledge Engineering*, pp. 582-589, 1992. [CrossRef] [Google Scholar] [Publisher Link]