

Representation of Dynamic Basic Block in Software Evolution Using Incidence Matrix

Rajeeb S. Bal*

Department of Computer Application, Maharaja Sriram Chandra Bhanja Deo University, 757003, India

E-mail: rajiv.s.bal@gmail.com

ORCID iD: <https://orcid.org/0000-0002-3030-5417>

*Corresponding author

Jibendu K. Mantri

Department of Computer Application, Maharaja Sriram Chandra Bhanja Deo University, 757003, India

E-mail: jkmantri@gmail.com

ORCID iD: <https://orcid.org/0000-0002-0039-5810>

Received: 17 October 2025; Revised: 05 December 2025; Accepted: 24 February 2026; Published: 08 April 2026

Abstract: Software evolution is a continuous process that transforms changing user requirements into improved software systems. Establishing a clear and well-structured development process is widely recognized as an effective means to enhance software maintainability, quality, and productivity. Tailoring software processes from existing process patterns and standards is essential for improving process performance, ensuring product quality, reducing development risks, and minimizing rework. Despite its importance, current research lacks a systematic and formally grounded method for tailoring software evolution processes. In this paper, we propose a structured approach based on Petri Net (PN) theory to address this limitation. There are four fundamental process constructs: sequence, concurrency, selection, and iteration are identified as basic building blocks for modeling software evolution processes. Using these constructs, four tailoring operations, namely adding, deleting, splitting, and merging, are formally defined. We study on the scalable process composition, matrix-based representations of Petri Nets (PNs) are employed. Incidence and related matrices provide a concise and mathematically tractable representation of both place/transition nets and restricted PNs, enabling the identification of essential structural properties of software processes. Also, we prove the reachability analysis and firing rules are utilized to derive a mathematical behavioral notation that captures binary relationships between input and output variables. This notation facilitates precise analysis of dynamic behavior for systematic software process tailoring.

Index Terms: Software Process, Software Evolution Process, Tailoring, Basic Block, Petri Net, Incidence Matrix.

1. Introduction

The term software was popularized in 1942 by John Tukey, but the discipline of software engineering began to take form only in the 1960s, when the growing complexity of software revealed the need for structured development practices. Margaret Hamilton used the term “software engineering” in 1963–1964 during her work on the Semi-Automatic Ground Environment (SAGE) system at the Draper Laboratory, emphasizing the necessity of engineering principles for developing reliable software. Later, in 1966, Anthony Oettinger used the term to highlight the difference between theoretical computer science and the practical challenges of designing and maintaining large, software-intensive systems. However, the field gained formal recognition at the 1968 NATO Software Engineering Conference, where Friedrich Bauer provided one of the first official definitions, marking the foundation of software engineering as an independent scientific and engineering discipline [1].

Figure 1. illustrates Yau’s evolutionary model of software development, depicting a staged and iterative framework for managing software evolution, emphasizing that software systems continuously change over time rather than being developed linearly. The process begins with a change request, which may originate from bug reports, new functional requirements, performance improvements, environmental factors, or user feedback. Each request is evaluated for feasibility, relevance, and potential impact, ensuring that unnecessary or risky changes are filtered out before further action. Once a request is accepted, the system enters the analysis and planning phase, where developers perform program comprehension to understand the system’s architecture, design, data flows, and interdependencies. Change impact analysis identifies all components, modules, and interfaces affected by the proposed change, allowing for an informed

assessment of risks, effort, and implementation strategy. The outcome is a structured plan that defines scope, resources, and potential side effects. During the implementation phase, approved changes are applied systematically. This may include restructuring or refactoring code to improve maintainability and modularity while preserving existing functionality. Change propagation ensures that all dependent modules, databases, and interfaces are updated consistently, minimizing ripple effects that could compromise system integrity.

After implementation, the system undergoes verification and validation. Verification confirms that the change has been correctly implemented according to design specifications, while validation ensures that the system meets functional and non-functional requirements. Testing at this stage may reveal defects or unintended consequences, which can trigger further iterations of analysis, planning, or implementation. The iterative feedback loops shown in the figure highlight the cyclic nature of software evolution, where each modification can generate new insights or additional change requests, resulting in repeated mini-cycles. By integrating change management, impact assessment, propagation, and validation into a continuous process, the model addresses system-wide dependencies, reduces errors, and ensures consistency, reliability, and maintainability of the software system over its lifecycle. This structured approach is particularly valuable for long-lived, complex software systems, where uncontrolled changes could otherwise lead to degradation, technical debt, or system failure.

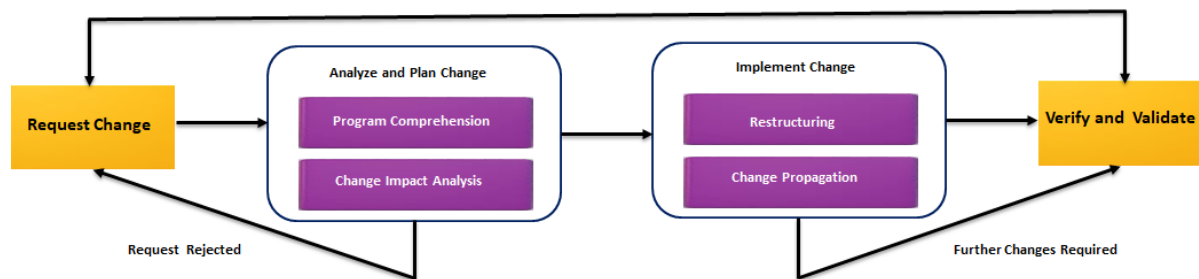


Fig.1. The organized process model for software evolution

Figure 2. explains the staged process model, which describes how a software system evolves over time as it ages. Software aging is natural and unavoidable, and the model illustrates the different stages a system passes through during its lifetime. The process begins with initial development, where the first working version of the software is created. After deployment, the system enters the evolution stage. In this stage, different types of changes and enhancements can be made, as long as the system's architecture remains stable and well-structured. As time passes, continuous modifications may damage the system's architecture. When the software becomes hard to modify without increasing complexity, it loses its ability to evolve easily. This is known as software decay, and the system moves into the servicing stage. During servicing, only small patches and minor fixes are applied to keep the system operational. If even small fixes become too expensive or difficult, the system enters the phase-out stage. In this stage, the system is still used but receives little or no maintenance. Eventually, the system reaches the close-down stage, where it is completely retired. If the system is still valuable to users despite its poor quality, it is called a legacy system. In such cases, it is better to plan a migration to a new system that provides similar or improved functionality. This planning should ideally start during the servicing stage to reduce risks and costs [2].

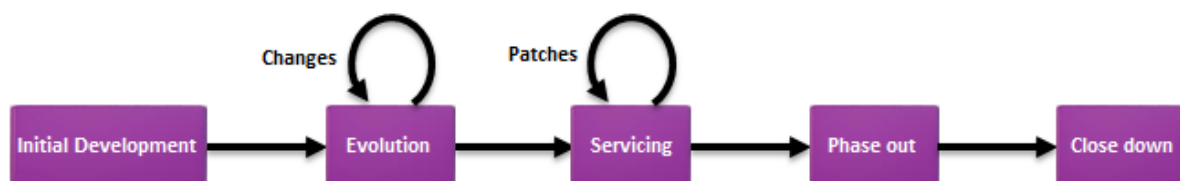


Fig.2. A staged model showing the evolution and aging of a software system

Software evolution refers to the ongoing refinement, adaptation, and enhancement of software systems using systematic methods, tools, and techniques. It encompasses several categories of changes: Perfective changes: improving performance, adding new features, or enhancing system usability to satisfy user or managerial needs. Corrective changes: identifying and repairing defects, errors, or faulty behavior. Adaptive changes: modifying software to operate in new environments, platforms, or hardware configurations. Preventive changes: restructuring or optimizing internal components to improve maintainability and reduce the likelihood of future problems. These types of changes rely on three critical tasks concerning, analyzing, and verifying the software which ensure that modifications preserve system quality, correctness, and operational integrity throughout the software lifecycle [3,4].

Despite decades of progress, tailoring the software evolution process remains a major challenge. Every project has unique constraints, requirements, and organizational environments, and there is limited understanding of how such characteristics influence the tailoring of evolution workflows. To address this gap, researchers have proposed the use of process reference blocks and tailoring rules to guide practitioners in customizing evolution processes according to project-

specific conditions [5]. These mechanisms offer structured guidance, enabling more flexible and accurate adaptation of workflows. However, there is still a need for more refined models that can formally represent, analyze, and validate tailored software evolution processes, especially in environments where changes occur rapidly and unpredictably.

PNs are a formal, mathematical, and graphical modeling framework designed to describe and analyze the behavior of discrete-event systems. They provide an effective means of modeling complex system dynamics involving concurrency, synchronization, resource sharing, sequencing, and conflicts among system activities. Introduced by Carl Adam Petri in 1962, PN combines a precise mathematical foundation with an intuitive graphical representation, making them suitable for both theoretical analysis and practical system modeling. Within software engineering, PN is extensively applied to the specification, modeling, and documentation of software systems and development processes, particularly those characterized by concurrent and distributed execution. The executable nature of PN enables simulation and visualization of system behavior, allowing designers to observe state changes and interactions during execution. Moreover, PN supports rigorous validation and verification techniques, including reachability analysis, deadlock detection, and performance evaluation. These capabilities make PN a powerful tool for analyzing correctness, reliability, and efficiency in complex software systems.

A PN is a directed bipartite graph composed of two types of nodes: places and transitions. Transitions are connected to input and output places via directed arcs and fire only when all input places contain at least one token. Tokens represent elements such as data items, conditions, or resources within the system. When a transition fires, it removes tokens from its input places and deposits tokens in its output places, thereby altering the marking of the net. The dynamic behavior of a PN is studied through these markings, where a state is reachable from another if there exists a legal sequence of transition firings connecting them. A state is considered live if any transition can eventually fire through some sequence, and bounded if the number of tokens in each place does not exceed a fixed integer across all reachable states; a net is safe if no place contains more than one token. During execution, conflicts may occur when multiple transitions compete for tokens from the same input place, which are typically resolved automatically according to PN semantics. For structural analysis, PN can also be represented in a compact, formal manner using matrix-based representations, particularly the incidence matrix. The incidence matrix encodes the connectivity between places and transitions along with the corresponding arc weights, providing a complete and fixed description of the net's structure. This representation is independent of behavioral properties such as liveness or boundedness and remains unchanged unless the net structure is modified. By combining the graphical PN model with its incidence matrix representation, both the structural and dynamic behavior of complex systems can be formally analyzed, simulated, and verified, making PN a powerful tool for modeling and studying concurrent and distributed software systems.

This paper aims to present a structured and organized model for software evolution, illustrated in Figure 1. and Figure 2., which outlines key tasks and activities involved in software change. The model defines the various stages of change, including the analysis, planning, implementation, and verification of changes. Each stage is interconnected, forming a cycle that helps guide the evolution of software systems in a controlled and predictable manner. By applying this model, software engineers can systematically adapt and improve software systems, ensuring that they meet the evolving requirements of users and maintain high standards of quality.

Contribution: This work contributes a structured and formal method for tailoring software evolution processes. It addresses the lack of systematic approaches in existing research by introducing a process modeling framework based on PN theory. The study identifies four fundamental process constructs: sequence, concurrency, selection, and iteration as the basic elements for representing software evolution activities. Building on these constructs, four tailoring operations are defined to support process customization, including adding, deleting, splitting, and merging process components. In addition, the work applies matrix-based representations of PN to support scalable process composition. By using incidence and related matrices, the proposed approach enables a compact and mathematically clear representation of software processes, making it easier to analyze their structural properties. The use of reachability analysis and firing rules further allows the derivation of a precise behavioral representation of software evolution processes. This supports accurate analysis of process behavior and provides a solid foundation for systematic and reliable software process tailoring.

The remainder of this article is organized as follows. Section 2 reviews related work relevant to software evolution and process modeling. Section 3 presents the study of PN structures and their matrix-based representations. In Section 4, the proposed software evolution process based on PN theory is introduced. Section 5 analyzes the adoption of the proposed approach in practice. Finally, Section 6 discusses the main findings of the study and presents the conclusions.

2. Related Works

In [6], we have studied that continuous software changes have become a normal part of business operations in the 21st century. The study also discusses the software development life cycle, current technological advancements, and the evolution of software maintenance. Additionally, it provides an overview of related research on re-engineering and reverse engineering techniques that support software evolution. Recent work further highlights the increasing role of automation and AI-assisted tools in managing complex system changes, enabling faster adaptation and improved maintainability. Moreover, researchers emphasize the growing importance of socio-technical factors, such as developer collaboration and knowledge sharing, in ensuring successful software evolution. In [7], a historical overview of the field is presented, highlighting four major research areas: program comprehension, re-engineering, reverse engineering, and

software repository mining. The study also reports key findings, summarizes important results, and identifies several challenges that remain open for future research in software evolution. Recent literature additionally stresses the need for more scalable analysis techniques and improved tool support to handle modern, large-scale software ecosystems, as well as the importance of integrating human factors into future evolution research. In [8], a categorization of various approaches that use patterns to understand, predict, plan, and monitor software evolution is presented. The categorization consists of two types: predictive analysis, conducted before evolution, and retrospective analysis, conducted after evolution has occurred. Predictive analysis focuses on identifying critical, sensitive, and vulnerable parts of the software, while retrospective analysis involves subdividing the software and its processes for evaluation. Recent studies also highlight the growing use of machine learning-based pattern detection to enhance both predictive and retrospective analyses, enabling more accurate forecasts of system behavior and deeper insights into post-evolution quality. In [9], it is argued that without properly considering requirements, it is difficult to determine which modification activities to undertake. Requirements serve as a bridge between the problem domain and the implementation domain, acting as an interface between the perspectives of stakeholders and developers. Recent work further emphasizes that continuously updated and well-managed requirements reduce the risk of incorrect modifications and support more aligned, value-driven software evolution. In [10], the use of clustering in software is discussed as a method for analysing software evolution. The study suggests that strategies originally developed for reorganizing software structure can be adapted to identify evolutionary trends and prevent harmful maintenance in the future. These strategies are applied to industrial software systems using conventional clustering techniques to study software evolution patterns. Recent research also explores more advanced clustering methods, such as density-based and hierarchical algorithms, to capture deeper structural changes and provide earlier warnings of potential degradation. In [11], a systematic analysis is presented to support research on software evolution visualization (SEV). The study explores the question: “What maintenance activities do current SEV technologies explicitly suggest, and how do they differ from one another?” Given the complexity of software evolution, there is a need for multi-criteria, comprehensive, and policy-driven SEV approaches to address existing software engineering objectives. In software research, it is essential to implement a tool for each visualization technique presented. Recent studies further highlight the importance of integrating interactive dashboards, real-time analytics, and immersive visual environments to improve developer understanding and decision-making during complex evolution tasks. In [12], a hybrid PN approach is applied to model the software development process. Routine tasks, such as collaborative work and interruptions caused by changes during development, are considered. Hybrid PNs are shown to be suitable for modeling and simulating the software development process effectively. Recent research also highlights the use of hybrid PNs for predicting bottlenecks, optimizing task scheduling, and supporting decision-making in dynamic software projects. In [13], the Sixth International Software Process Workshop (ISPW6) was held in Hakodate, Japan, from October 29–31, 1990. This workshop focused on various aspects of software processes. A combined model representing the software development process as a sequence of distinct steps executed on a continuously changing project platform was presented. The viability of this model was demonstrated by merging a specific event model of the ISPW6 development process with the dynamic system structure developed by AbdelHamid and Madnick. This combined modeling approach enables managers to examine issues and newly discovered problems effectively. Furthermore, this approach has inspired later research on adaptive process modeling and simulation frameworks that can handle evolving project requirements and dynamic team interactions. In [14], architecture-based reliability models are discussed, including the foundational and component PNs. For component-based software architectures, a time-extended PN model is proposed, which separates transitions into timed and immediate transitions. The model introduces time-delay reliability and temporal reliability for timed transitions, enhancing PNs’ ability to represent real-time integrated software systems. Recent studies extend this model to incorporate stochastic behaviors and probabilistic fault predictions, improving reliability assessment for complex, distributed software systems. In [15], software dynamic evolution is analyzed, emphasizing its major impact on mission reliability and the difficulty of computation. The study addresses three key questions regarding software mission reliability: how software architecture depends on task decomposition, how task paths define evolution rules, and how reliability is affected during software evolution. Recent research further explores automated analysis techniques and simulation tools to predict reliability impacts more efficiently, helping developers manage evolution in complex, mission-critical systems. In [16], PNs are applied to study parallel and distributed discrete-event dynamic systems. The formalism allows timed modeling, where real events are related through synchronization and sharing phenomena. The paper surveys key topics in PN models, highlighting the importance of logical and structural perspectives and discussing connections between queuing networks and PN models. Recent studies extend PN applications to model large-scale distributed software systems, incorporating stochastic timing and resource constraints to improve performance analysis and reliability prediction. In [17], a comprehensive review of PN research is presented, emphasizing place/transition systems and PN theory applications. The study notes a growing interest in advanced PN types, such as timed nets, stochastic nets, and high-level nets, which allow modeling of temporal, probabilistic, and hierarchical behaviors. These developments extend the applications of PNs across multiple domains, including manufacturing, software engineering, communication networks, and workflow systems. In [18], a method for automated scenario analysis using PNs is proposed. This method supports correctness, consistency, and completeness checks, while enabling early identification of potential problems in software development. It also enhances traceability, linking issues in PNs back to their source scenarios, which improves software testing and validation processed. In [19], rule specifications for software process models are proposed, using the German Process Model (GV-Model) as an example. The GV-Model highlights errors in informal studies. Research is conducted to correct these errors and implement the model effectively. The study also proposes rules for tailoring software

process models, enabling the derivation of customized executable process models.

3. Petri Net Structure, Incidence Matrix

A PN is a bipartite graphical model used to represent discrete event systems, where places are depicted as circles and transitions as bars or rectangular boxes. Directed arcs connect places to transitions and transitions to places, indicating the flow of tokens and defining the dynamic behavior of the system. The initial marking is shown by tokens (black dots) placed in the places, representing the current state of the system. The firing of a transition occurs when all its input places contain the required number of tokens. When the transition fires, it consumes tokens from its input places and produces tokens in its output places, reflecting a state change in the system. This mechanism allows PNs to model concurrent, sequential, and parallel processes, as multiple transitions may fire simultaneously if their input conditions are satisfied. The graphical structure (showed in Figure 3) not only visualizes system states and transitions but also provides a formal framework for analyzing reachability, liveness, and system behavior. PNs are therefore widely used for modeling, simulation, and verification of complex systems in software engineering, manufacturing, workflow management, and communication networks [20-25].

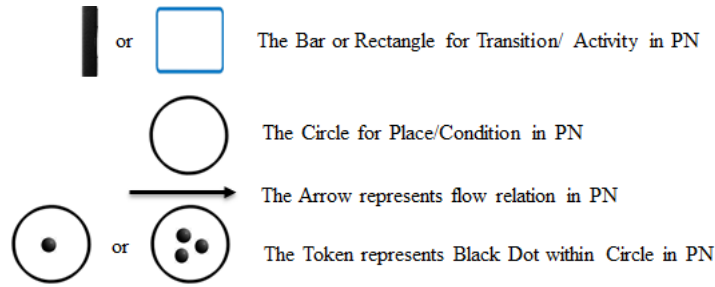


Fig.3. The Elements or Components of Graphical Representation of a PN

Definition of a Petri Net with Initial Marking: A Petri net is a five-tuple, $PN = (P, T, F, W, M_0)$, where, $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places or conditions, $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of transitions or activities, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation), $W: f \rightarrow \{1, 2, 3, \dots\}$ is a weight function, $M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$ is the initial marking, $P \cap T = \emptyset$ and $P \cup T = \emptyset$.

A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted by N . A Petri net with the given initial marking is denoted by (N, M_0) .

In PN, a node directs to either a place or a transition, x is a node, if and only if $x \in (P \cup T)$. The pre-set or input set of transition t implies the set of all input places to t . It can be written as $\cdot t = \{p \in P \cap I(p, t) \neq 0\}$. The post-set or output set of transition t implies the set of all output places from t . It can be written as $t \cdot = \{p \in P \cap O(p, t) \neq 0\}$. A straightforward path is identified as a order of nodes in PN $\{a_1, a_2, \dots, a_n\}$ where $n > 1$ and $\exists_{arc}(a_i, a_{i+1})$ for $i \in N_{n-1}$ if $n > 1$ and $a_i = a_j \Rightarrow i = j$ where $N_n = \{1, 2, \dots, n\}$ possibly determining a self-loop, simple a loop or a circuit.

The Representation of Incidence Matrix: A concise way to present Petri nets structurally for study is to represent PNs using matrices form. The incidence matrix is a constant i.e. it does not change for the PN unless PN structure is altered. The incidence matrix captures all the detail about a PN structure i.e. the flows from place to transition or transition to place with all the number of transitions is represented in a form. Hence, the incidence matrix of PN structure is controlled of $C_{ij} = O_{ij} - I_{ij}$. We denoted C_{ij} by C which is collected of the difference between the matrix of output flow (O_{ij}) and input flow (I_{ij}). Hence, the incidence matrix can be represented as

$$C_{ij} = C_{ij}^+ - C_{ij}^- \quad (1)$$

where C_{ij}^+ denotes the output matrix of transition to place and C_{ij}^- denotes the input matrix of place and transition.

Transition Enabling and Firing, State Equations: In PN, we consider m and n denote the set of places and transitions, respectively. Further, in tailoring software evolution process, set of places and transitions same as conditions and activities respectively. The incidence matrix of a PN can be represented by $C = [C_{ij}]_{n \times m}$. And its typical entry is given by

$$C_{ij} = C_{ij}^+ - C_{ij}^- \quad (2)$$

where C_{ij}^+ represents the number of flows from transition i to the output place j and C_{ij}^- represents the number of flows to transition i from the input place j .

In the PN structure, a marking (or a state) gives to each place a non-negative integer. If a marking gives to place p a non-negative integer k i.e., p has k tokens. Graphically, we place k dots in place p . In tailoring software evolution process, we use the concept of conditions and activities represent places and transitions. A transition (or an activity) has a certain

number of input and output places representing the pre-conditions and post-conditions of the condition, respectively. We simulate the dynamic behavior of a system; a token distribution or marking is changed as per the transition (firing) rule: i. A transition is said to be enabled if each input place of the transition is marked with at least one token. A fireable transition may be chosen to fire. ii. The transition (firing) consists of removing one token from each of its input places, and adding one token to each of its output places.

There are three characteristic of PNs. They are: reachability, liveness, and safeness (or boundedness). A final marking M_n is said to be reachable from an initial marking M_0 if there exists a firing sequence which transforms M_0 to M_n .

4. Software Evolution Process

Software Evolution in software engineering refers to the continuous process of developing, maintaining, and modifying software after its initial release to meet changing requirements, fix defects, or improve performance. Unlike the initial development phase, software evolution is ongoing and inevitable, as software must adapt to new technologies, user needs, and business environments. PNs provide a formal modeling framework that can effectively support software evolution activities by representing the dynamic behavior and state changes of software systems. Specifically [26,27,28,29,30]:

Program Comprehension: PNs can model software workflows, module interactions, and control flows, making it easier to understand complex system structures and trace dependencies between components.

Reverse Engineering: PNs can be derived from existing software to abstract its dynamic behavior. For example, state changes, concurrent processes, and event sequences can be represented in a PN, providing insight into system operation without modifying the code.

Reengineering: Using PNs, developers can simulate proposed changes to software, such as restructuring modules or optimizing workflows, to predict impacts before implementing them. This helps ensure correctness and minimize disruption.

Software Repository Mining: Historical evolution data from version control systems and bug trackers can be translated into PN models, showing how components and processes have evolved over time. This enables analysis of patterns, identification of bottlenecks, and validation of system evolution strategies.

A PN is a formal modeling framework consisting of places, transitions, a flow relation, and a marking that assigns tokens to places to represent system states. Places denote conditions or states, transitions represent events or activities, and the firing of transitions governs state changes. PN execution is inherently nondeterministic, as multiple transitions may be enabled simultaneously. Based on token interpretation, PNs can be classified into Boolean-based models, where each place holds at most one token and represents binary states, and integer-based models, where places may contain multiple tokens to represent resource quantities. While integer-based models offer greater expressive power, their execution semantics can be ambiguous in distributed settings, leading to the frequent adoption of restricted forms such as 1-safe PNs. In this work, Petri nets provide a formal foundation for modeling software evolution processes, enabling rigorous structural and behavioral analysis through matrix representations and state equations.

Definition of software process system: It is defined as a 4-tuple $\mathcal{E} = (C, A, F, M)$, where (C, A, F) is a net without isolated elements, and $C \cap A = \emptyset$. C is a finite set of conditions; each element $c \in C$ represents a condition of the software process. A is a finite set of activities; each element $a \in A$ represents an activity, and the occurrence of a is referred to as the execution or firing of the activity. $M \subseteq 2^C$ is called the case class of $\mathcal{E}$, where 2^C denotes the power set of C . Each element $m \in M$ represents a possible case or state of the software process. For every activity $a \in A$, there exists at least one case $m \in M$ such that a is enabled in m , meaning that the required conditions for executing a are satisfied.

Definition of software process : It can be defined by $p = (C, A, F, M_0)$ is an instance of a software process system $\mathcal{E} = (C, A, F, M)$, where M_0 is the initial marking defining the starting distribution of tokens in conditions, and the process evolves as activities fire according to F , moving tokens and updating the system state.

Formal Definition of Activity Enabling: An activity a is said to be enabled in a given marking M if and only if the following condition holds for all conditions $c \in C$:

$$M(c) \geq \text{Pre}(c, a) \quad (3)$$

where $M(c)$ denotes the number of tokens contained in condition c under marking M , and $\text{Pre}(c, a)$ represents the precondition of activity a with respect to condition c , i.e., the number of tokens required in condition c for activity a to execute. This condition guarantees that an activity can be executed only when all its required input conditions contain a sufficient number of tokens. Hence, activity execution is strictly controlled by the current marking of the system.

Flow Interpretation: If a flow (arc) exists from a condition c to an activity a , then $\text{Pre}(c, a) > 0$, indicating that tokens are consumed from condition c when activity a fires. Conversely, if no such flow exists, then $\text{Pre}(c, a) = 0$, meaning that condition c does not contribute tokens to activity a . Similarly, if a flow exists from activity a to a condition c , then $\text{Post}(a, c) > 0$, indicating that tokens are produced in condition c upon the firing of activity a . If no such flow exists, then $\text{Post}(a, c) = 0$.

Matrix Representation: The enabling condition can also be expressed using matrix notation. Let M be the marking vector and Pre be the precondition matrix, whose columns correspond to activities and whose rows correspond to conditions. Activity a is enabled if:

$$M \geq Pre(:, a) \quad (4)$$

where the inequality is evaluated element-wise. This formulation provides a compact and precise method for checking activity enablement and supports efficient analysis of system behavior using linear algebra.

Matrix Based Representation of Petri Net Dynamics: The behavior of a Petri net can be conveniently expressed using vectors and matrices, which allows precise mathematical modeling of the process evolution.

Let M_0 denote the initial marking, representing the number of tokens in each place of the net at the starting state. C denote the incidence matrix of the Petri net. Each entry C_{ij} indicates the net effect of transition j on place i (positive for tokens added, negative for tokens removed). f denote the firing vector, whose elements specify how many times each transition fires. The elements of f are non-negative integers, since a transition cannot fire a negative number of times.

The resulting marking M_1 after firing the transitions is given by:

$$M_1 = M_0 + C \cdot f \quad (5)$$

This equation captures the linear effect of firing: each transition moves tokens between places according to the incidence matrix, and the firing vector determines how many times each transition occurs.

By repeating this operation, a sequence of markings can be generated:

$$\{M_0, M_1, M_2, \dots, M_n\} \quad (6)$$

Each marking in the sequence represents the state of the net at a given step, showing how tokens are distributed in the places as transitions fire.

Furthermore, multiple firings can be collected in a matrix, where each row (or column) represents a firing vector at a particular step. This allows compact representation of the entire net evolution and makes analysis of reachability, concurrency, and resource usage straightforward. Transition firing in a Petri net is essentially a vector-based operation, reducible to matrix arithmetic, which provides a clear and computationally efficient way to model and analyze discrete-event systems.

Tailoring in Software Evolution refers to the customization of software evolution processes to fit project-specific requirements, constraints, and organizational guidelines. Rather than applying a generic evolution process, tailoring adapts methods, workflows, and tools to match the unique characteristics of a project, such as team size, software complexity, domain requirements, or risk factors. A tailored software evolution process offers several benefits:

Enhanced Work Efficiency: By customizing workflows and responsibilities, the team can focus on critical activities, avoid unnecessary steps, and reduce time and effort.

Quality Assurance: Tailoring ensures that standards, best practices, and validation checks are applied appropriately, which improves the quality of software products.

Risk Minimization: Project-specific adaptations allow early detection and management of potential issues or bottlenecks, reducing the likelihood of defects or project delays.

Reduced Rework: By aligning the evolution process with project goals and requirements, unnecessary modifications or corrections are minimized, saving resources.

Tailored processes are typically modeled using formal approaches, often based on an Evolution Process Meta-Model (EPMM). The EPMM extends PNs by integrating object-oriented concepts and Hoare logic, allowing precise representation of states, transitions, and constraints in the evolution process. This formalism helps in analyzing, simulating, and verifying software evolution activities. A four-level architecture of the software evolution process (illustrated in Figure 4) usually includes:

Meta-level: It defines rules and guidelines for evolution.

Process-level: It specifies coordinated software processes.

Project-level: It tailors processes for specific projects.

Instance-level: It Represents actual software evolution activities in real projects.

The basic block can be defined in the tailoring software evolution process as per EPMM. But a basic block can be converted to two ways of a PN structured: i. without initial marking, and ii. with initial marking.

Definition of Basic Block (B_B) without Initial Marking: The basic block denoted by B_B and represented by Petri Net structure without initial marking. The B_B can be described by 5-tuple. $B_B = (C, A, F, A_e, A_x)$, where $C = \{c_1, c_2, \dots, c_n\}$ is a finite set of conditions ($n \geq 0$), $A = \{a_1, a_2, \dots, a_m\}$ is a finite set of activities ($m \geq 0$), F is set of arcs or flow relation i.e., $F \subseteq (C \times A) \cup (A \times C)$, A_e is the entrance activity, A_x is the exit activity i.e., $A_e, A_x \subseteq A$.

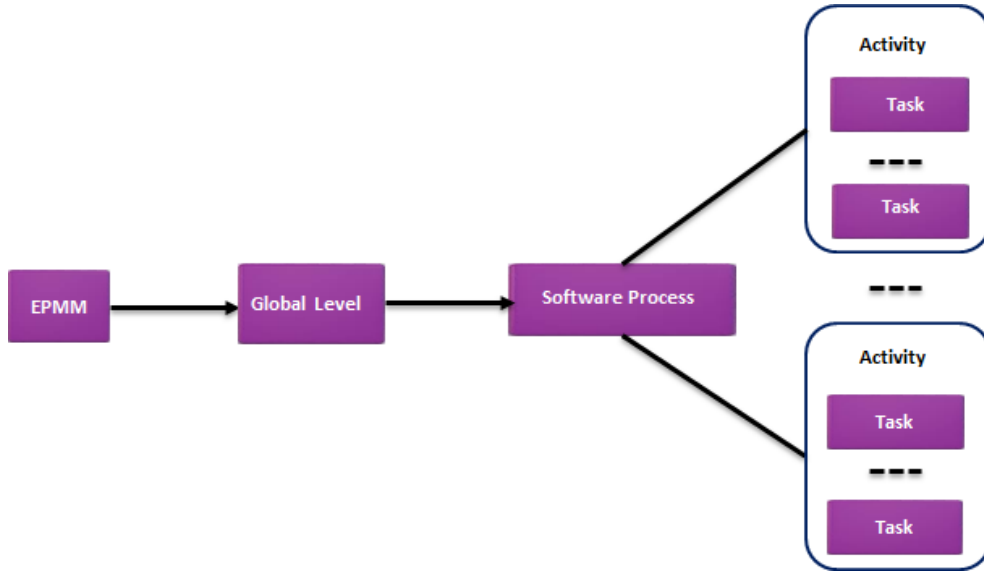


Fig.4. The four level architecture of software evolution process possesses

Definition of Basic Block (B_B) with Initial Marking: The basic block denoted by B_B and represented by Petri Net structure with initial marking. The B_B can be described by 7-tuple. $B_B = (C, A, F, A_e, A_x, W, M_0)$, where $C = \{c_1, c_2, \dots, c_n\}$ is a finite set of conditions ($n \geq 0$), $A = \{a_1, a_2, \dots, a_m\}$ is a finite set of activities ($m \geq 0$), F is set of arcs or flow relation i.e., $F \subseteq (C \times A) \cup (A \times C)$, A_e is the entrance activity, A_x is the exit activity i.e., $A_e, A_x \subseteq A$, $W: f \rightarrow \{1, 2, 3, \dots\}$ is the weight function, $M_0: P \rightarrow \{0, 1, 2, 3, \dots\}$ is the initial marking.

5. Result Analysis

We define four types of basic blocks (B_B) in tailoring software process, they are sequence block, concurrence (parallel) block, selection block, and iteration (loop) block. Each block has its own algorithm, and each basic block is extended into a PN structure.

Definition of Sequence Block: A Sequence Block is a construct that encloses activities executed one after another, ensuring strict sequential flow from the first activity to the last activity shown in figure 5.

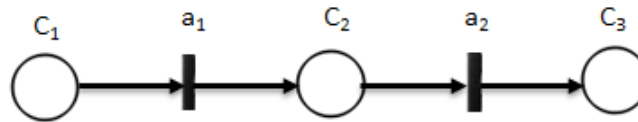


Fig.5. The sequence block without initial marking in software process

Here, we apply the definition of PN structured: without initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 5. From the figure 5, we get $C = \{c_1, c_2, c_3\}$, $A = \{a_1, a_2\}$, $F = \{(c_1, a_1), (a_1, c_2), (c_2, a_2), (a_2, c_3)\}$, $A_e = \{a_1\}$, $A_x = \{a_2\}$.

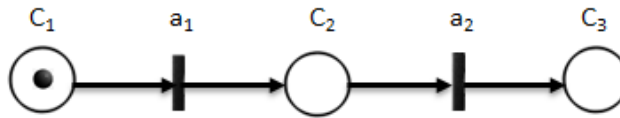


Fig.6. The sequence block with initial marking in software process

Similarly, we apply the definition of PN structured with initial marking. As per definition of PN structured without initial marking shown in figure 6. From the figure 6, we get $C = \{c_1, c_2, c_3\}$, $A = \{a_1, a_2\}$, $F = \{(c_1, a_1), (a_1, c_2), (c_2, a_2), (a_2, c_3)\}$, $A_e = \{a_1\}$, $A_x = \{a_2\}$, $M_0 = (1, 0, 0)$.

Using Equation (1) for Figure 5, we obtain the incidence matrix: $I_{ij} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $O_{ij} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ and $C_{ij} = O_{ij} - I_{ij} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix}$.

Again, using equations (3) and (4) for the sequence block shown in Figure 6, the software evolution process is tailored as follows. From the figure 6, we get the initial marking $M_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$. The activities or transitions a_1 and a_2 are

enabled because $M_0 = Pre(\cdot, a_1) \Rightarrow Pre(\cdot, a_2) = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow M_0 = Pre(\cdot, a_1) \Rightarrow M_0 > Pre(\cdot, a_2)$.

Here, we apply equations (5) and get dynamic behavior, enabling conditions, and reachability graph. Now, the result of dynamic behavior as follows. From the figure 6, there is one initial marking in C_1 and none elsewhere: $M_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$,

Incidence matrix $C = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$, fire a_1 once using equation (5), we get $M_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$. Similarly, if

fire a_2 once then $f = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, $M_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$. After firing a_1 and a_2 once, we get the final result of dynamic

behavior $M = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$. Finally, we get reachability graph of sequence block of software process shown in figure 7.

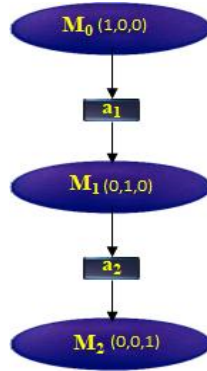


Fig.7. The reachability graph of sequence block of software process

Definition of concurrence block: A Concurrency Block encloses activities that are executed simultaneously, allowing parallel and independent execution shown in figure 8.

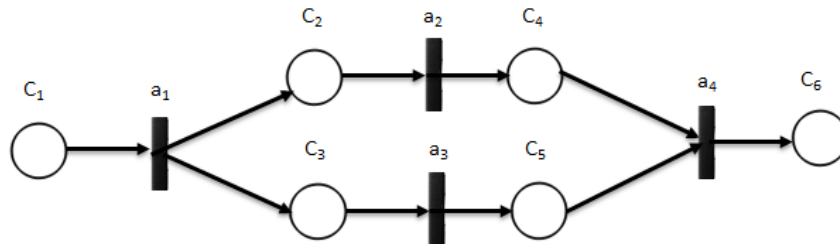


Fig.8. The Concurrency or parallel block without initial marking in software process

Here, we apply the definition of PN structured without initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 8. From the figure 8, we get $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, $F = \{(c_1, a_1), (a_1, c_2), (a_1, c_3), (c_2, a_2), (a_2, c_4), (c_4, a_4), (c_3, a_3), (a_3, c_5), (c_5, a_4), (a_4, c_6)\}$, $A_e = \{a_1\}$, $A_x = \{a_6\}$.

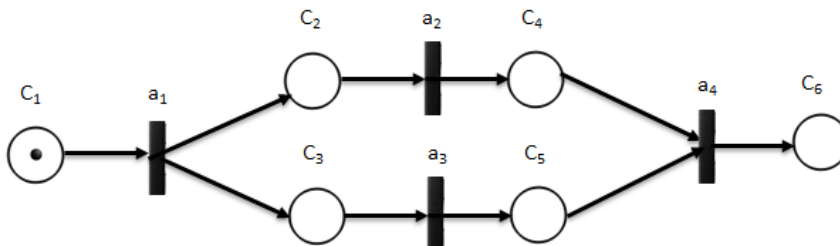


Fig.9. The Concurrency or parallel block with initial marking in software process

Similarly, we apply the definition of PN structured with initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 9. From the figure 9, we get $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$, $F = \{(c_1, a_1), (a_1, c_2), (a_1, c_3), (c_2, a_2), (a_2, c_4), (c_4, a_4), (c_3, a_3), (a_3, c_5), (c_5, a_4), (a_4, c_6)\}$, $A_e = \{a_1\}$, $A_x = \{a_6\}$, $M_0 = (1, 0, 0, 0, 0, 0)$.

Using Equation (1) for Figure 9, we obtain the incidence matrix: $I_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$, $O_{ij} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ and

$$C_{ij} = O_{ij} - I_{ij} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
 Again, using equations (3) and (4) for the sequence block shown in figure 8, the

software evolution process is tailored as follows. From the figure 9, we get the initial marking $M_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$. The activities

or transitions a_1 and a_2 are enabled because $M_0 = Pre(\cdot, a_1)$, $Pre(\cdot, a_2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $Pre(\cdot, a_3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $Pre(\cdot, a_4) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

and $Pre(\cdot, a_4) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow M_0 = Pre(\cdot, t_1)$, $M_0 > Pre(\cdot, t_2)$, $M_0 > Pre(\cdot, t_3)$ and $M_0 > Pre(\cdot, t_4)$.

Here, we apply equations (5) and get dynamic behavior, enabling conditions, and reachability graph. Now, the result

of dynamic behavior as follows. From the figure 9, there is one initial marking in C_1 and none elsewhere: $M_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

Incidence matrix $C = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, fire a_1 using equation (5), we get $f = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, $M_1 = M_0 + Cf = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$,

Similarly, Fire a_2 or a_3 (concurrent), if fire t_2 , $M_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, fire t_3 , $M_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$, parallel behavior exists. Fire the remaining

one from either marking, $M_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$. After firing a_4 , we get the final result of dynamic behavior of concurrency or parallel

block of software process $M_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$. Finally, we get reachability graph of concurrency or parallel block of software process

shown in figure 10.

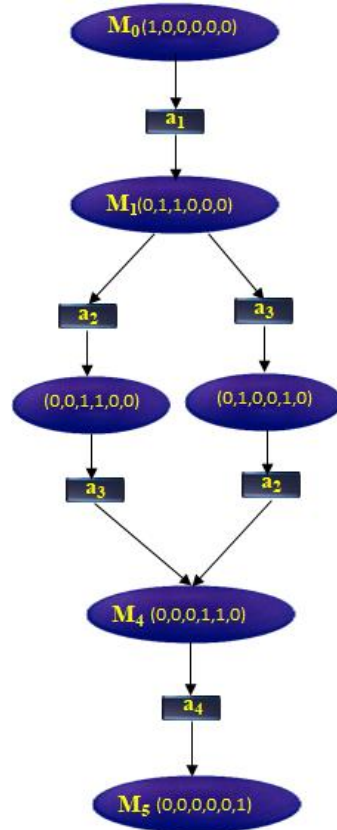


Fig.10. The reachability graph of concurrence or parallel block of software process

Definition of selection block: A Selection Block encloses activities where only one activity is chosen and executed, based on conditions or decisions shown in figure 11.

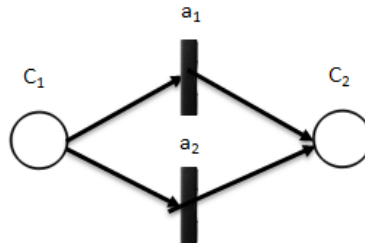


Fig.11. The selection block without initial marking in software process

Here, we apply the definition of PN structured without initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 11. From the figure 11, we get $C = \{c_1, c_2\}$, $A = \{a_1, a_2\}$, $F = \{(c_1, a_1), (c_1, a_2), (a_1, c_2), (a_2, c_2)\}$, $A_e = \{a_1, a_2\}$, $A_x = \{a_1, a_2\}$.

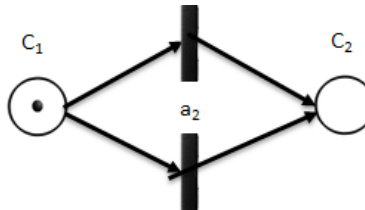


Fig.12. The selection block with initial marking in software process

Similarly, we apply the definition of PN structured with initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 12. From the figure 12, we get $C = \{c_1, c_2\}$, $A = \{a_1, a_2\}$, $F = \{(c_1, a_1), (c_1, a_2), (a_1, c_2), (a_2, c_2)\}$, $A_e = \{a_1, a_2\}$, $A_x = \{a_1, a_2\}$, $M_0 = (1,0)$.

Using Equation (1) for Figure 11, we obtain the incidence matrix: $I_{ij} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$, $O_{ij} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$ and $C_{ij} = O_{ij} - I_{ij} = \begin{bmatrix} 0-1 & 0-1 \\ 1-0 & 1-0 \end{bmatrix} = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$.

Again, using equations (3) and (4) for the sequence block shown in Figure 12, the software evolution process is tailored as follows. From the figure 12, we get the initial marking $M_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. The activities or transitions a_1 and a_2 are enabled because $M_0 = Pre(\cdot, a_1)$, $Pre(\cdot, a_2) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \Rightarrow M_0 = Pre(\cdot, a_1)$, $M_0 > Pre(\cdot, a_2)$.

Here, we apply equations (5) and get dynamic behavior, enabling conditions, and reachability graph. Now, the result of dynamic behavior as follows. From the figure 12, there is one initial marking in C_1 , none in and none C_2 , $M_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, incidence matrix $= \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$, fire a_1 using equation (5), we get $ff = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $M_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Similarly, fire a_2 $f = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $M_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Hence both firings lead to the same marking. For New enabling condition at M_1 , $M_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, if a_1 needs 1 then in c_1 is not enabled, if a_2 needs 1 then in c_1 is not enabled. Finally, we get reachability graph of selection or choice block of software process shown in figure 13. Hence, both transitions lead from M_0 to the same marking M_1 .

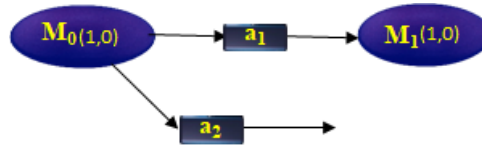


Fig.13. The reachability graph of selection or choice block of software process

Definition of Iteration or Loop Block: The execution of activities is concurrently is known as the repeatedly block shown in figure 14.

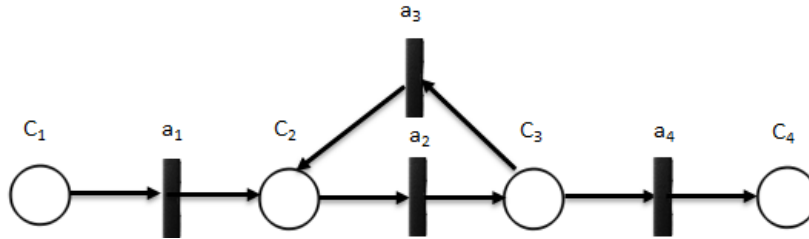


Fig.14. The iteration or loop block without initial marking in software process

Here, we apply the definition of PN structured without initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 14. From the figure 14, we get $C = \{c_1, c_2, c_3, c_4\}$, $A = \{a_1, a_2, a_3, a_4\}$, $F = \{(c_1, a_1), (a_1, c_2), (c_2, a_2), (a_2, c_3), (c_3, a_3), (a_3, c_2), (c_3, a_4), (a_4, p_4)\}$, $A_e = \{a_1\}$, $A_x = \{a_4\}$.

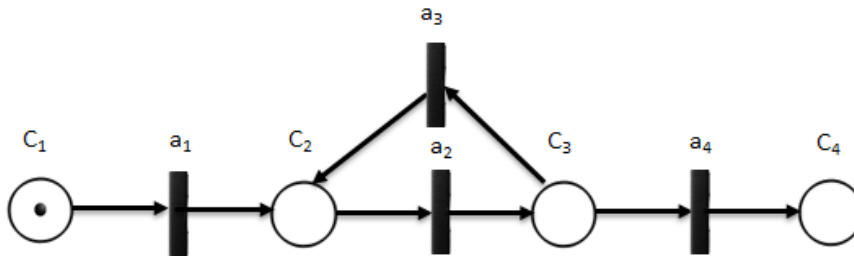


Fig.15. The iteration or loop block with initial marking in software process

Similarly, we apply the definition of PN structured with initial marking, and with initial marking. As per definition of PN structured without initial marking shown in figure 15. From the figure 15, we get $C = \{c_1, c_2, c_3, c_4\}$, $A = \{a_1, a_2, a_3, a_4\}$, $F = \{(c_1, a_1), (a_1, c_2), (c_2, a_2), (a_2, c_3), (c_3, a_3), (a_3, c_2), (c_3, a_4), (a_4, p_4)\}$, $A_e = \{a_1\}$, $A_x = \{a_4\}$, $M_0 = (1, 0, 0, 0)$.

Using Equation (1) for Figure 14, we obtain the incidence matrix: $I_{ij} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$,

$$O_{ij} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } C_{ij} = O_{ij} - I_{ij} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Again, using equations (3) and (4) for the sequence block shown in Figure 12, the software evolution process is

tailored as follows. From the figure 15, we get the initial marking $M_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$. The activities or transitions a_1 and a_2 are

enabled because $M_0 = Pre(\cdot, a_1)$, $Pre(\cdot, a_2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, $Pre(\cdot, a_3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ and $Pre(\cdot, a_4) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \Rightarrow M_0 = Pre(\cdot, a_1)$,

$M_0 > Pre(\cdot, a_2)$, $M_0 > Pre(\cdot, a_3)$ and $M_0 > Pre(\cdot, a_4)$.

Here, we apply equations (5) and get dynamic behavior, enabling conditions, and reachability graph. Now, the result of dynamic behavior as follows. From the figure 15, there is one initial marking in C_1 and none elsewhere: $M_0 = (1, 0, 0, 0)$,

Incidence matrix $C = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, fire a_1 using equation (5), we get $f = (1, 0, 0, 0)^T$,

$M_1 = M_0 + Cf = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$, Similarly, Fire a_2 , $f = (0, 1, 0, 0)^T$, $M_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$. Now, the

$M_2 = (0, 0, 1, 0)$ enabled activities or transitions a_3 and a_4 under conflict. As per conflict, we create two cases: Case A:

fire a_3 (loop), $f = (0, 0, 1, 0)^T$, $M_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ and returns to c_2 . Case B: fire a_4 (exit), $f = (0, 0, 0, 1)^T$,

$M'_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$. Hence, M'_3 is enabled transitions of dead marking.

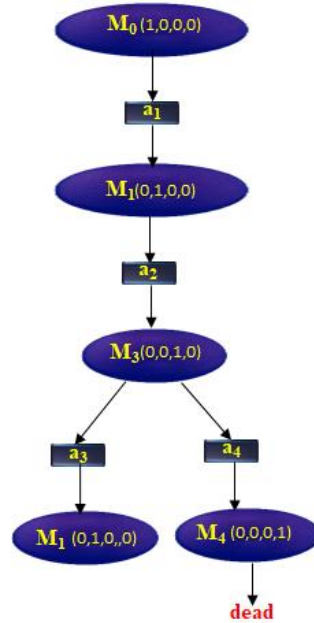


Fig.16. The reachability graph of iteration or loop block of software process

Addition Operation of Basic Blocks in Software Process: The PN model shown in figure 17 illustrates the composition of two fundamental software process behaviors using the addition operation. The left sub-net represents a concurrency-oriented process fragment, while the right sub-net models an iterative execution fragment. Each fragment is independently well-structured and captures a specific aspect of dynamic software behavior, such as parallel task execution or controlled repetition. The addition operation combines these basic Petri-net structures by integrating their places, transitions, and flow relations into a single unified net, while preserving the internal execution semantics of each fragment.

Through this operation, the resulting model represents a more complex software process in which concurrent activities may be followed by iterative processing stages. This modular composition supports incremental modeling and reuse of verified process fragments, which is essential in large-scale software process engineering. However, as illustrated in the composed net, the addition operation introduces interactions between transitions belonging to different behavioral fragments. In particular, shared intermediate places enable multiple transitions simultaneously, leading to competition for tokens. This situation results in structural conflicts and nondeterministic execution, since the firing of one transition may disable others that are enabled at the same time. Such behavior is undesirable in software processes that require predictable control flow and deterministic execution. Therefore, while the addition operation is a powerful mechanism for modeling dynamic and composite software behavior, it must be augmented with appropriate control conditions or synchronization constraints to eliminate conflicts and ensure correct process execution.

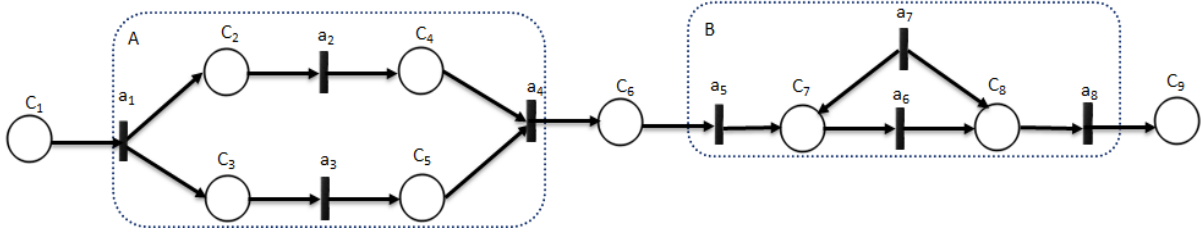


Fig.17. The addition operation of basic blocks in software process using sequence structure

In figure 17, we consider two blocks one is Concurrency block i.e. A and another are loop or iteration block i.e. B. After adding operation, we get block C. But there is a conflict at the time of adding. Thus, the conflict is avoided by condition are as follows:

- $A.a_e = B.a_x$.
- $A.a_e$ is a part of $B.a_x$.

Similarly, we can use the definition of B_B , the conditions of enabling and firing for concurrency or parallel block, selection block and loop or iteration block in tailoring software evolution process. Now, we apply the completely controllable or completely reachable for adding operation of heterogeneous block in the software evolution process shown in figure 18 initial marking.

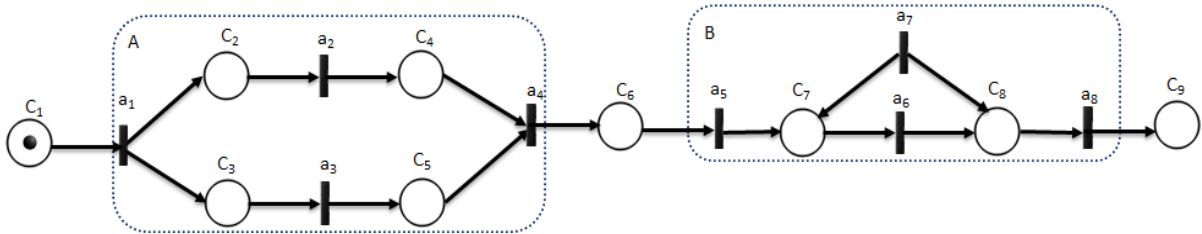


Fig.18. Note how the caption is centered in the column. The sequence block with initial marking at condition c_1 using firing

In the figure 16, we find $A.a_e = a_1$, $A.a_x = a_4$, $B.a_e = a_5$, $B.a_x = a_8$ with initial state denoted by $M_0 = (1,0,0,0,0,0,0)^T$. In the state M_0 , transition a_1 is firable. After firing of a_1 we calculate M_1 with token i.e.,

$$M_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

After firing of a_2 with state $M_1 = (0,1,1,0,0,0,0,0)^T$ and we get $M_2 = (0,0,0,1,0,0,0,0)^T$. Similarly, after firing of a_3 with state $M_1 = (0,1,1,0,0,0,0,0)^T$ and we get $M_3 = (0,0,0,0,1,0,0,0)$ and after firing of a_4 with state $M_3 = (0,0,0,0,1,0,0,0)^T$ and we get $M_4 = (0,0,0,0,0,1,0,0,0)^T$ and shown in Figure 19.

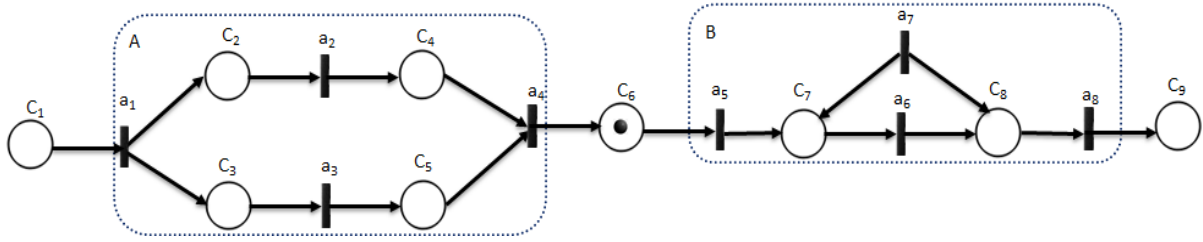


Fig.19. The activity firing at a_4 with marking at c_6 for adding operation of heterogeneous block in software evolution process

After firing of a_5 with state $M_4 = (0,0,0,0,0,1,0,0,0)^T$ and we get $M_5 = (0,0,0,0,0,0,1,0,0)^T$. Similarly, after firing of a_6 with state $M_5 = (0,0,0,0,0,0,1,0,0)^T$ and we get $M_6 = (0,0,0,0,0,0,0,1,0)^T$ and after firing of a_7 with state $M_6 = (0,0,0,0,0,0,0,1,0)^T$ and we get $M_7 = (0,0,0,0,0,0,0,1,0,0)$. Hence, after firing of a_8 with state $M_6 = (0,0,0,0,0,0,0,1,0)^T$ and we get $M_8 = (0,0,0,0,0,0,0,0,1)^T$ shown in figure 20. The controllability and reachability concept of dynamic systems is applied to basic blocks for addition operation in tailoring software evolution process.

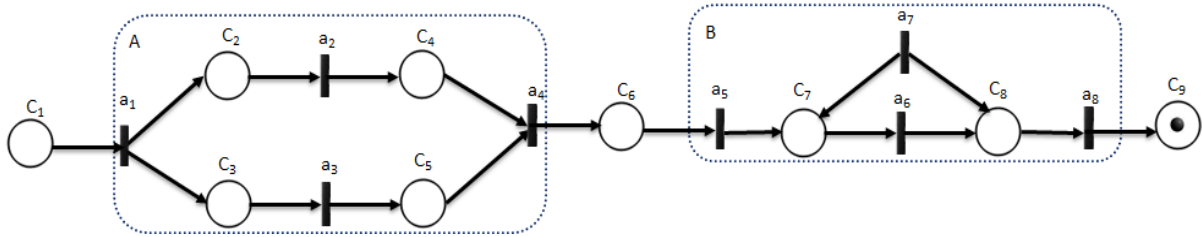


Fig.20. The activity firing at a_8 with marking at c_8 for completely controllable or completely reachable in adding operation for heterogenous block

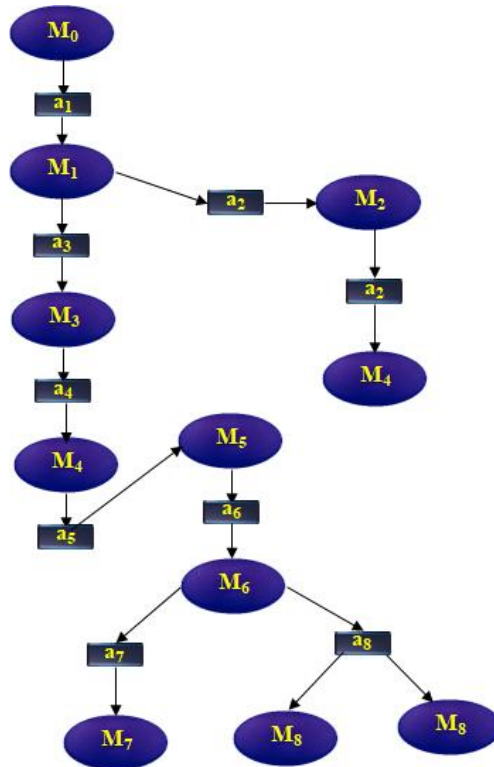


Fig.21. The reachability graph of adding operation for heterogenous block

Here, from the figure 21, the final markings M_8 or M_9 represent completion of the extended software process after the addition operation.

6. Conclusions

This paper has presented a systematic and formal approach to tailoring software evolution processes based on four fundamental behavioral constructs: sequence, concurrency, selection, and iteration. These constructs are modeled using an extended Petri-net structure, which provides a precise and executable representation of dynamic software process

behavior. The use of Petri nets enables the explicit modeling of control flow, synchronization, and iteration, which are essential characteristics of evolving software processes. In formal analysis, the PN model is transformed into an incidence matrix representation. The incidence matrix offers a concise and mathematically rigorous description of the structural relationships between places and transitions, allowing key properties of the tailored process to be examined. In particular, the matrix-based state equation enables systematic analysis of transition firing sequences and marking evolution, thereby facilitating verification of structural consistency and process correctness after tailoring operations such as addition. Furthermore, the reachability graph derived from the incidence matrix and firing rules provides a complete representation of all reachable states of the tailored process. The analysis demonstrates that the process obtained through the addition of basic blocks is fully reachable and controllable, ensuring that all intended execution states can be achieved without introducing deadlocks or unintended behavior. These results confirm that the proposed approach enables effective management, analysis, and verification of tailored software evolution processes through formal Petri-net-based techniques.

In future, it will focus on extending the proposed Petri-net-based software process tailoring framework in several directions. Timed and stochastic Petri nets will be explored to incorporate temporal constraints and uncertainty into software evolution modeling. Tool support will be developed to automate tailoring operations, including adding, deleting, splitting, and merging, while preserving interface consistency across abstraction levels. Furthermore, resource, cost, and effort parameters will be integrated into the model to enable quantitative evaluation and optimization of software processes. Formal verification techniques, such as model checking, will be applied to ensure properties including reachability, liveness, and deadlock freedom. Finally, the approach will be validated through comprehensive industrial case studies to assess scalability and practical applicability.

Author Contributions Statement

Rajeeb S. Bal - Conceptualization, Writing – original draft.
Jibendu K. Mantri - Writing – review and editing.

All authors have read and agreed to the published version of the manuscript.

Conflict of Interest Statement

The authors declare no conflicts of interest.

Funding Declaration

Not applicable.

Data Availability Statement

No datasets

Ethical Declarations

Not applicable.

Acknowledgments

We sincerely thank the reviewers and domain experts for their valuable comments and constructive suggestions. Their insightful feedback has significantly enhanced the technical quality, clarity, and overall reliability of this work.

Declaration of Generative AI in Scholarly Writing

Generative AI tools were used solely for language refinement and readability enhancement, while all research content and interpretations were developed and verified by the authors, who assume full responsibility for the manuscript.

Abbreviations

The following abbreviations are used in this manuscript:

PN - Petri Net
PNs - Petri Nets
SAGE - Semi Automatic Ground Environment
GV Model - German Process Model

References

- [1] Grady Booch, The History of Software Engineering, IEEE, 2018. DOI: 10.1109/MS.2018.3571234.
- [2] Tom Mens, Serge Demeyer, Software Evolution, Springer-Verlag Berlin Heidelberg, 2008. DOI: 10.1007/978-3-540-76440-3.

- [3] Miryung Kim, Na Meng, Tianyi Zhang, Software Evolution, Springer Nature Switzerland AG, 2019. <https://doi.org/10.1007/978-3-030-00262-6>.
- [4] Hongji Yang, Martin Ward, Successful Evolution of Software Systems, ARTECH HOUSE, INC., 2003.
- [5] Peng Xu, Balasubramaniam Ramesh, Process management Using Process Tailoring to Manage Software Development Challenges, IEEE, 2008. DOI: 10.1109/MITP.2008.81.
- [6] Hongji Yang, Martin Ward, Successful Evolution of Software Systems, ARTECH HOUSE, INC., 2003.
- [7] Andy Zaidman, Martin Pinzger, Arie van Deursen, Software Evolution, Encyclopedia of Software Engineering, 2010.
- [8] Tom Mens, Serge Demeyer, Future Trends in Software Evolution Metrics, IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution, ACM, 2001. DOI: <https://doi.org/10.1145/602461.602476>.
- [9] Neil A. Ernst, Alexander Borgida, John Mylopoulos, Requirements Evolution Drives Software Evolution, IWPSEEVOL '11: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, ACM, 2011. DOI: <https://doi.org/10.1145/2024445.2024450>.
- [10] John Davey, Elizabeth Burd, Clustering and Concept Analysis for Software Evolution, IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution, ACM, 2002. DOI: <https://doi.org/10.1145/602461.602494>.
- [11] Renato Lima Novais, Andre Torres, Thiago Souto Mendes, Manoel Mendonça, Nico Zazworka, Software evolution visualization: A systematic mapping study, Elsevier, 2013. DOI: <http://dx.doi.org/10.1016/j.infsof.2013.05.008>.
- [12] Bo XIE, Xiaojun LI, Hybrid petri nets modeling for software development process management, 2nd IEEE International Conference on Computer Science and Information Technology, IEEE, 2009. DOI: 10.1109/ICCSIT.2009.5234890.
- [13] Robert H. Martin, David Raffo, A Model of the Software Development Process Using Both Continuous and Discrete Models, SOFTWARE PROCESS IMPROVEMENT AND PRACTICE, John Wiley & Sons, Ltd. 2000. DOI: [https://doi.org/10.1002/1099-1670\(200006/09\)5:2/3;1-7](https://doi.org/10.1002/1099-1670(200006/09)5:2/3;1-7); AID-SPIP122;3.0.CO;2-T.
- [14] Chi Zhang, Yunyun Ma, Xiaohua Wang, Ruixue Wang, Software Architecture Modeling and Reliability Evaluation Based on Petri Net, Fourth International Conference on Dependable Systems and Their Applications, IEEE, 2017. DOI: 10.1109/DSA.2017.17.
- [15] Yumei Wu, Yongli Yu, The Mission Reliability Modeling Methodology for Software Dynamic Evolution, International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE), IEEE, 2013. DOI: 10.1109/QR2MSE.2013.6625599.
- [16] Manuel Silva, Javier Campos, PERFORMANCE MODELS BASED ON PETRI NETS, Proceedings of the IMACS/IFAC Second International Symposium on Mathematical, 1993.
- [17] Tadao Murata, Petri Nets: Properties, Analysis and Applications, Proceedings of the IEEE, IEEE, 1989. DOI: 10.1109/5.24143.
- [18] Edgar Sarmiento, Julio Cesar Sampaio Do Prado Leite, Eduardo Almentero, Analysis of Scenarios with Petri-Net Models, 29th Brazilian Symposium on Software Engineering, IEEE, 2015. DOI: 10.1109/SBES.2015.13.
- [19] D. Welzel, H.L. Hausen, W. Schmidt, Tailoring and conformance testing of software processes: the ProcePT approach, Proceedings of Software Engineering Standards Symposium, IEEE, 1995. DOI: 10.1109/SESS.1995.525950.
- [20] Rozenberg, G., Thiagarajan, P.S. (1986). Petri nets: Basic notions, structure, behaviour., In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds) Current Trends in Concurrency. Lecture Notes in Computer Science, vol 224, Springer, 2005. DOI: <https://doi.org/10.1007/BFb0027048>.
- [21] Petri Nets - Applications, Y Narahari, General Article, 1999.
- [22] Gajendra Pratap Singh, APPLICATIONS OF PETRI NETS IN ELECTRICAL, ELECTRONICS AND OPTIMIZATIONS, International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), IEEE, 2016. DOI: 10.1109/ICEEOT.2016.7755078.
- [23] ANTHONY (TONY) SPITERI STAINES, Matrix Representations for Ordinary Restricted Place Transition Nets, WSEAS TRANSACTIONS on COMPUTERS, 2017.
- [24] Anthony Spiteri Staines, Ordinary Petri Net Matrices, ITM Web of Conferences 24, AMCSE, 2018. DOI: <https://doi.org/10.1051/itmconf/20192402007>.
- [25] Anthony Spiteri Staines, Some Fundamental Properties of Petri Nets, International Journal of Electronics Communication and Computer Engineering, IJECCE, 2013.
- [26] Fei Dai, Tong Li, Tailoring Software Evolution Process, Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, IEEE, 2007. DOI: 10.1109/SNPD.2007.25.
- [27] Li Tong, An Approach to Modelling Software Evolution Processes, Springer-Verlag Berlin Heidelberg, 2009. DOI: <https://doi.org/10.1007/978-3-540-79464-6>.
- [28] Lauren H. Quinn, Mark P. Ginsberg, Process Tailoring and the Software Capability Maturity Model, Software Engineering Institute, Technical Report, CMU/SEI-94-TR-024 ESC-TR-94-024, Carnegie Mellon University Pittsburgh, PA 15213, 1995.
- [29] Fei Dai, Tong Li, Composing Software Evolution Process Component, Advanced Parallel Processing Technologies. Lecture Notes in Computer Science, vol 4847. Springer, Berlin, Heidelberg, APPT, 2007. DOI: https://doi.org/10.1007/978-3-540-76837-1_74.
- [30] Andy Zaidman, Martin Pinzger, Arie van Deursen, Software Evolution, Auerbach Publications, 2010.

Authors' Profiles



Rajeeb S. Bal is a PhD Research scholar in the Department of Computer Applications, The Maharaja Sriram Chandra Bhanja Deo University, Baripada, Odisha, India. He was working as Assistance Professor in the Department of Computer Science and Engineering of Ajay Binay institute of Technology at Cuttack, Odisha, India. He received his MCA degree from F. M. University and M.Tech(CSE) from Biju Patnaik University of Technology, Odisha. His research interests include Software Engineering, Wireless Sensor Network, Soft Computing.



Prof. Jibendu K. Mantri, M.Sc, M.Phil, M.Tech, Ph.D, Former Professor, Deptt.Computer Application, MSCB University, is working as a Professor, School of Information System ASBM University, India, now. He has more than 33years of experience of teaching and research. His area of specialization includes: AI, Business Process Re-Engineering and Computer Security. He has authored 6 books in total. He has also authored more than 117 papers published in refereed International / National Journals and Conferences. He is an active reviewer for peer-reviewed journals and has delivered invited talks pertaining to his research field.

How to cite this paper: Rajeeb S. Bal, Jibendu K. Mantri, "Representation of Dynamic Basic Block in Software Evolution Using Incidence Matrix", International Journal of Information Technology and Computer Science(IJITCS), Vol.18, No.2, pp.65-82, 2026. DOI:10.5815/ijitcs.2026.02.05