A Study on Petri Net Based Model-Driven Development in Software Evolution

(ソフトウェア派生開発におけるペトリネットに基づくモデル駆動開発に関する研究)

March 2017

Mohd Anuaruddin Bin Ahmadon

Graduate School of Science and Engineering,
Yamaguchi University

Abstract

Software needs to adapt to new function, thus upgrade and patches are required. As programmers apply patches and upgrades to its source code, new version is released based on its product line. Hence, when releasing new version, it is important to verify if the software follows the specification in the product line. Managing product line is important in software evolution; major and minor upgrades are frequently applied to meet requirements for adding new functions, improve performance or strengthen security features. In conventional software development, a program is developed through many stages which we call as the stage of evolution. Throughout each cycles of software release, software becomes larger and complex. However, in order to satisfy product line where certain functions must be preserved, we must verify the backward compatibility. In conventional ways, backward compatibility is verified by looking at the source code or running the program. This approach consumes a lot of time and effort because backward compatibility must be verified for each release. One formal analysis approach is by analyzing the program's behavior by model checking. However, model checking approach enumerates all states which is intractable for large and complex software. This is due to well-known state space explosion problem. Moreover, conventional development approach do not consider to preserve the backward compatibility in software evolution.

In this paper, we proposed a model-driven development in software evolution. The thesis is organized as follows:
In Chap. 1, we gave the background, motivation and objective of the research. We also stated the position of the research by discussing the related work.

In Chap. 2, we introduced the fundamentals knowledge of software evolution, Petri nets and other important properties and theory.

In Chap. 3, we introduced the real world concept of software evolution. As an example, we take a smart refrigerator derivative development where we upgrade the functions and improve the performance of the embedded software. Then we gave the problem in software evolution known as backward compatibility. For a software X, does the newer version software Y preserve impor-

tant functions in software X? We formalized the problem and proposed a theorem for the relation of backward compatibility and behavioral inheritance. Then, we proposed two conditions that satisfies backward compatibility known as behavioral inheritance and response property.

In Chap. 4, we proposed a solution for the development that preserves backward compatibility throughout the evolution. We utilized Petri net for the approach. First, we reverse engineer a progam into Petri net model. Then, we can modify a program based on its model or verify whether backward compatibility is preserved or not. Then, we proposed a backward compatibility verification that checks behavioral inheritance and response property. We showed that our method avoids state space explosion. We only need to verify the Petri net's structure and we utilized a tree representation called as process tree as the solution. We showed that we can verify behavioral inheritance to verify backward compatibility for minor version upgrade and response property analysis for major version upgrade.

In Chap. 5, we showed the tool development by showing an application example of software evolution.

In Chap. 6, we gave the conclusion and future works of the development method in the software evolution.

要旨

題目: ソフトウェア派生開発におけるペトリネットに基づくモデル駆動開発に関する研究

ソフトウェアの長寿命化や大規模化、短納期化に伴い、新規開発の機会は減少し、現行のソースコードを変更したり、新しい機能だけをコーディングして追加したりする派生開発が一般的になっている．派生開発では、機能の追加や変更が繰り返され、ソフトウェアが「つぎはぎ」だらけになることも少なくない．その結果、変更漏れや変更間違い、変更前後の互換性の損失などの深刻な問題が発生している．これらの問題は派生開発に特有なものであり、従来の開発法では解決が困難である．したがって、派生開発に適した新たな開発法の確立が望まれている．

本研究ではソフトウェア派生開発での支援を提案するためにソフトウェア解析問題の計算複雑さを明らかにした．ソフトウェアに対してプロダクトラインに従うことが重要であるが、派生開発では様々な機能がアップグレードされ、プログラムがますます複雑になって行く．また、プロダクトラインに従うために重要な機能を維持することが重視され、互換性の解析が注目されている．従来の開発方法では互換性の維持されない場合があり、アップグレードの際に互換性の検証を繰り返さなければならない．そのため、開発には時間や手間がかかる．本研究では、互換性が維持される開発方法を明らかにした．アプローチとしてモデル駆動開発方法に注目する．モデルに基づくプログラムの開発が一般的になっているが、本研究ではさらにプログラムからモデルに変換し、互換性を検証してから新バージョンのプログラムを開発する方法を提案した．

本研究ではリバースエンジニアリングの方法に基づいてプログラムをペトリネットモデルに変換し、モデル検証を行う．まず、C プログラムをペトリネットに変換するアルゴリズムを提案した．プログラムをネットモデルに変換してから、解析が可能になる．しかし、大規模で複雑なプログラムに対して、ソフトウェアの互換性を解析するために、従来の方法で網羅的に状態を列挙すると状態空間爆発が起こり、解析が困難になった．そのため、互換性の検証に関して、状態空間爆発を回避できる方法を明らかにした．

第 1 章では研究の目的と背景を説明する．そして、本研究の位置を示すために関連研究

を紹介する．

第2章でソフトウェア派生開発、ペトリネットの基礎と重要な性質を紹介する．

第3章ではソフトウェア派生開発の現実世界における問題とその解決法の概要を紹介する．ソフトウェア派生開発における問題を二つに分けた．1つ目の問題はリバースエンジニアリングの問題で、2つ目の問題は互換性の検証問題である．この2 つの問題に注目して、解決法を明らかにした．まず、リバースエンジニアリングの方法を示す．プログラムからペトリネットモデルに変換するツールC2PNMLを提案した．また、アプローチを示すために現実世界のスマート冷蔵庫の派生開発を例として挙げた．

第4章では互換性の問題を解決するための全体的なソフトウェア派生開発の方法を提案する．互換性の検証では、マイナーアップグレードとメジャーアップグレードに分けて、検証を行う．マイナーアップグレードではプログラムの振る舞い継承の検証が必要である．一方、メジャーアップグレードでは応答性質の検証が必要である．本研究では状態を網羅的に列挙しない振る舞い継承の解析法と応答性質の解析法を明らかにした．振る舞い継承の検証はプログラムの違いをチェックするためにペトリネットの構造を解析する方法である．また、応答性質の解析はプログラムの大事な命令順序関係をチェックする方法である．

第5章では、ツールの開発を紹介し、評価結果を明らかにした．最後に、第6章では結論と今後の課題を説明する．

本研究の新規性はソフトウェア派生開発にモデル駆動開発に基づくアプローチを提案した．ソフトウェア派生開発では互換性問題を振舞い継承と応答性質問題に分けてそれぞれが満たされれば互換性が維持される開発法を明らかにした．振舞い継承にはハンドルというペトリネットの構造をチェックする．また、応答性質にはプロセスツリーという表現バイアスを用いて、木のノードを解析することで命令順序関係を表現する．全体的なアプローチとしてリバースエンジニアリングの方法で、プログラムをモデル化してから、互換性をチェックする．

本研究の有効性として、互換性の検証では状態空間爆発を回避する方法を提案した．振る舞い継承の解析にはネットの構造を解析するだけで、状態空間の違いが分かり、状態を列挙する必要はない．また、プロセスツリーという表現バイアスを用いることでプログラムの命令順序関係を木探索アルゴリズムにより多項式時間で解析できる．従来のモデルチェッカーでは状態を全て列挙する必要があるので、大規模のプログラムには手に負えなくなる．プロセスツリーでは状態数はプロセスツリーのパターンによって決まる．そのため、大規模のプログラムに対して100万以上の状態を実装時間で計算できる．また、大規模で複雑なプログラムに対して命令順序関係を検証するには命令を一つずつ実行しなくても、プロセスツリー上で深さ優先探索で検証することが可能になった．

Acknowledgements

I would like to express my gratitude to my supervisor Assoc. Prof. Dr. Shingo Yamaguchi for his invaluable guidance, support and motivation in this research work. Also to the professors in the department for examining this thesis including Prof. Dr. Shinya Matsufuji, Prof. Dr. Masanao Obayashi, Prof. Dr. Qi-Wei Ge and Assoc. Prof. Dr. Yoshinobu Tamura. I also would like to thanks all professors in the Graduate School of Science and Engineering for their direct and indirect support.

Part of this thesis was also made successful with the support and advice from Dr. Brij Bhooshan Gupta from National Institute of Technology Kurukshetra, India as the visiting and fellow researcher in our laboratory. I also would like to thank Prof. Dr. Tomohiro Hase from Ryukoku University, IEEE Fellow, the founder and director of Global Conference of Consumer Electronics (GCCE) for the chance and encouragement to actively involved in the IEEE CE Society. I also would like to express my appreciation to all students in the Software System Laboratory for their support and contributions to my research work.

I also would like to thank IEICE, IEEE and Inderscience Publisher for reviewing and publishing my conference papers and journals. Particularly, to the IEEE Consumer Electronics Society for awarding me the CE Society East Japan Chapter ICCE 2016 Young Scientist Paper Award in the United States and GCCE 2016 Outstanding Student Paper Award in Kyoto.

This research was also partially supported by Interface Corporation, Japan. I also would like to thank MARA Education Foundation for granting me the doctoral scholarship under the Malaysia-Japan Higher Education (MJHEP) Program and also my appreciation to Japan University Consortium for Transnational-education (JUCTe) for their support during my study period. Also, my gratitude to Yamaguchi University for granting me the financial support from 50th Anniversary Project Grant that enabled me to present my research overseas and for the support that allowed me to join the Global Engineer Training Program in India and Malaysia.

Finally, I am deeply grateful especially to my wife Siti Aisyah for her invaluable sacrifice and understanding me the best as a doctorate student. Without her dedicated love, support, motivation and continuously to pray for my success through this agonizing period and her patience in facing hardships together, I will not be able to successfully complete this doctorate thesis in the most positive and motivated way. Also, a very special thanks to my parents Ahmadon Abu and Siti Hasrah Abd. Samad, my parents-in-law Mokhtar Ramli and Rosnani Ahmad for their blessing, prayer and moral support throughout all these years. Also, special thanks to all my family members for their support in my life.

List of Symbols

$x \in X$: x is an element of set X.

$X \subset Y$: Set X is contained in set Y.

$X \cup Y$: Union of sets X and Y.

$X \cap Y$: Intersection of sets X and Y.

$X \times Y$: Products of sets X and Y.

$X - Y$: Difference of sets X and Y.

$\emptyset$: Empty set.

$\mathbb{N}$: The set of natural numbers.

$|X|$: The cardinality of set X.

Contents

Abstract **iii**

Acknowledgements **vii**

List of Symbols **viii**

1 Introduction **3**

1.1 Background and Motivation of the Thesis 3

1.2 Research Objective 4

1.3 Related Work 5

1.4 Novelty and Effectiveness 7

1.5 Organization of the Thesis 7

2 Preliminary **9**

2.1 Software Evolution 9

2.2 Petri Net 9

2.3 Workflow Net 10

2.4 ST-net 11

2.5 Soundness 12

2.6 Process Tree 12

2.7 Behavioral Inheritance 14

2.8 Backward Compatibility 15

2.9 UML Activity Diagram 16

3 Software Evolution and Petri Net Approach **17**

3.1 Real World Concept of Software Evolution 17
3.1.1 Derivative Development 18
3.1.2 Real World Example 21
3.2 Problem in Software Evolution 25
3.2.1 Backward Compatibility in Software Evolution 25
3.2.2 Backward Compatibility Verification Problem 27
3.3 Our Approach 34
3.4 Translation of Program Structure into Petri Net 34
3.5 Representation of Program Structure with WF-net 42
3.5.1 Decision Method for Bridge-less of WF-nets 45
3.5.2 Necessary and Sufficient Condition 45
3.5.3 Polynomial-time Procedure 46
3.5.4 Example 47
3.6 Remarks 49

4 Model-Driven Development in Software Evolution 51
4.1 Convertibility of Workflow Net to Process Tree 52
4.1.1 Convertibility problem 53
4.1.2 Necessary and Sufficient Condition 53
4.1.3 Conversion Algorithm of Workflow Net to Process Tree 58
4.1.4 Example of Conversion 61
4.2 State Number Calculation 63
4.2.1 State Number Calculation Problem and Its Properties 64
4.2.2 Polynomial Time Procedure 68
4.2.3 Application Example 71
4.3 Behavioral Inheritance Analysis 74
4.4 Response Property Analysis 77
4.4.1 Decidability and Complexity of Response Property 80
4.4.2 Polynomial Time Procedure 83
4.4.3 Application Example 88
4.5 Remarks 92

5 Other Application and Evaluation **93**

5.1 Security Protocol Implementation and Its Verification 93

5.2 Process Tree Analysis Tool 99

5.3 Remarks 102

6 Conclusion **103**

Bibliography **105**

Chapter 1

Introduction

1.1 Background and Motivation of the Thesis

Nowadays, software development method is rapidly changing. There are many software development methodologies introduced such as waterfall development, agile development, spiral development and scrum development [1]. In this development processes, throughout the software life-cycle, product family line management is important. Software is developed based on the requirements and new releases always include new functionalities, extensions or interfaces. In this situation, the newer version of the software depends on the scale of the changes that are affected by minor or major upgrades. Despite of minor or major upgrade in new software releases, the underlying program code changes. Hence, the software itself can derailed from its product family line specifications. This can cause confusion to the software users and vendor.

When releasing new version, it is important to verify the core functionality of the old version in the new version. In conventional software development, a program evolves as developers make changes to its source code. In order to verify the product family line, the developers need to verify the code and execute the program themselves. This validation process is repeated at each development stages. The problem is that the process costs a lot of time and effort for large and complex program. This situation applies to large scale embedded program where derivative development takes place [2, 3].

A program is developed through many stages which we call as the stage of evolution [4]. During the development stage, we need to write, compile and debug a program based on its design diagram i.e. UML [5] and flowchart. Generally, the program should follow its design diagram that defines its behavior. Nowadays, most compilers are made to identify grammatical and syntax errors before compilation. Although the program passes the compilation, it depends on the programmer to verify the behavior of the program. For example, for a grammatically

correct program that passed a compiler check with statement A, B and C, we need to verify whether (i) statement A are executed in sequence with B and C; and (ii) B and C are executed in parallel. In this case, although compiler does not displays any grammatical or syntax errors, a programmer still needs to check for correctness of the executions between statements. Does the behavior of the program satisfy its design model? In conventional ways, verification by looking at the source code or running the program consumes a lot of time and effort. One formal analysis approach is by analyzing the program's state space. However, state space explosion may occur for large and complex programs.

1.2 Research Objective

The objective of this research is to propose two key methods to support model-driven development [6]; (i) translation of program into Petri net method; and (ii) behavioral analysis method. In the beginning of development we need to convert program into Petri net models so it can be used for behavioral analysis. Then, we can analyse the converted Petri net with our behavioral analysis method. We utilize a representational bias called as process tree. Our approach enables conversion from source code to Petri net model. Then the model can be used for verification of execution sequence that should be preserved since the original model was developed. For example, we have a software model of version 1. The model is then being developed into a program. Let us say that we need to develop an upgraded version the program, but we need to preserve some important parts. First, we upgrade or patch the program into a version 2. After modifying the original source code, we can verify the program's important structures by translating it into the model of version 2 and compare it to the model of version 1. Our methods play an important role in the translation and verification procedure. We will explain the details of our approach in the next section.

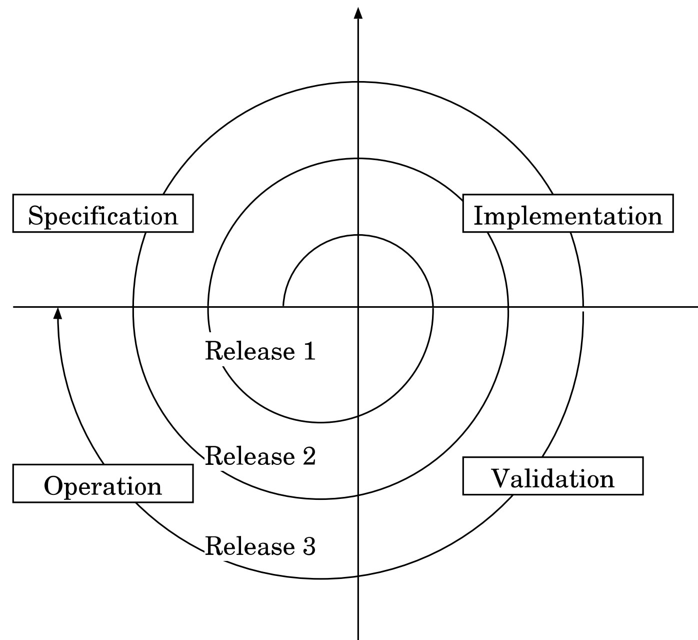


Figure 1.1: Development stages in software evolution.

1.3 Related Work

There are several approaches proposed for the design and verification of program in order to follow the product family line specifications. An example of an implementation of a program specification is security protocol. Designing new protocol implementation is a very challenging task which has high probability of generating errors. Recently, new techniques have been introduced to improve specification implementation such as protection profile for operating systems [7] and software-defined specifications for mobile systems [8]. These techniques require validation on the implementation of their specifications. Moreover, the method of patchwork approach is adopted to adapt to evolution and revolution of technology growth due to features and security requirements in software and systems [9].

Therefore, formal verifications for security protocols is very important. One of the approach for verification of security protocols is based on high-label abstract formal model analysis [10]. Approaches that apply the high-level abstract model involves extracting the security model from the program which was implemented from formal protocol description. Moffet et al. [11] proposed a verification on software behavior using model checking by state space exploration of the software. Fu et al. [12] proposed a model-based formal verification by extending the Input/Output Label Transition System (IOLTS for short). They defined a non-negligible security properties as the part of the transition system and analysed the probable transition sequence of the IOLTS.

Another work was done by modeling security protocol with UML. Smith et al. [13] proposed a method to design security protocol with abstract model UML2 and its verification procedures to detect potential security flaws. Similar work was also done by Goubault-Larrecq et al. [14] and Chaki et al. [15]. In these methods, however, there was no method which to confirm whether the security protocols are implemented into a program based on its specifications. The approaches are only limited to vulnerability analysis of security properties in the specifications but not on the successful implementations of the protocol itself.

It is hardly known that whether we can verify the protocol implementation inside the software itself. Petri net [16] is a graphical and mathematical analysis tool for the analysis of software behavior. Generally, Petri net can represent program structure. Rana et al. [17] proposed a method for performance analysis of Java program using object-oriented Petri net. In order to apply Petri net analysis techniques, we need to translate programs into Petri nets. Voron et al. [18] stated that translating a program to Petri net in a large scale requires a lot of effort and time. So a tool to automatically translate a program to Petri net is needed. Voron et al. proposed a translation tool called as Evinrude [19] but the resulting Petri net is a general Petri net where perspectives must be specified which are for particular program only.

Once a program is translated in Petri net model, we can analyze its behavior. A well-known method for behavioral analysis of Petri net is by using model checking tool [20, 21]. Model checking approach is exhaustive. However, exhaustive approach costs a lot of computation time because we need to enumerate all behavioral states. Another model checking approach for large systems is by slicing Petri nets into subnets and run the model checking processes in parallel [22]. However, the slicing technique is restricted to certain subclasses of Petri net only. Other than model checking approach, an algorithm utilizing representational bias called as process tree [23] to avoid state space explosion was proposed. We proposed a convertibility check [24] and state number calculation problem based on process tree [25]. In this research, we utilize process tree in our software analysis approach.

As for related work regarding to software evolution, Erdweg et al. [26] proposed the usage of extensible languages SugarJ into source code to check the partial behavior changes in programs. Oyetoyan et al. [27] proposed a detection method of component dependencies during software evolution. Toyoshima et al. [28] proposed the application of refactorization method for changing software code that preserved its behavior. We proposed a security protocol implementation verification method in software evolution [29]. The method utilized behavioral inheritance notion to verify backward compatibility of programs.

1.4 Novelty and Effectiveness

The novelty of this research are, we propose a model-driven development approach based on Petri net to tackle backward compatibility problem in software evolution. We proposed two major methods; (i) Reverse engineering method for translation of parallel structured program into Petri net; and (ii) We propose model-driven verification method two verify two important properties for backward compatibility problem; (i) Behavioral inheritance problem for minor upgrade and (ii) Response property analysis for major upgrade. We also propose two tools to support the model-driven development in software evolution. Our first tool is C2PNML which reverse engineer parallel structured program into Petri net. The output Petri net model can be used for analysis. Our second tool is Process Tree Analysis Tool to analyze process tree for the purpose of state number calculation and response property analysis.

The effectiveness of this research is we can verify backward compatibility of programs with behavioral inheritance and response property. The method verify and guarantee if behavioral inheritance and response property is preserved, then the backward compatibility is preserved in the evolution. Our method is polynomial and can avoid state space explosion problem and be run in practical time compared to model checking. Our method simply analyze the structure of Petri net called as handles to verify the behavioral inheritance and use process tree to fast check statements execution with tree search algorithm. Therefore, state space explosion can be avoided without enumerating all states in the program. Our method support overall program behavior analysis compared to the previous work.

1.5 Organization of the Thesis

This thesis is organized as follows:

In Chap. 1, we gave the background, motivation and objective of the research. We also stated the position of the research by discussing the previous work.

In Chap. 2, we introduce the fundamentals knowledge of software evolution, Petri nets and other important properties.

In Chap. 3, we introduce the real world concept of software evolution, its problem and real world example. Then, we show the overview of our approach. We show that in order to analyze software evolution, we need to model the program into Petri net. In this chapter, we proposed a reverse engineering method by translating a software program into a Petri net model. We propose an algorithm to translate C program into Petri net model and revealed the class of Petri net that

can represent general program structure.

In Chap. 4, we proposed a model-driven verification method by proposing program analysis with state number calculation, behavioral inheritance and process tree. The proposed method can be used to verify backward compatibility of program in software evolutions. We proposed polynomial time solution for all methods and show the application to real world example.

In Chap. 5, we showed the tool development and application example of software evolution. we proposed a tool for process tree analysis and evaluated the tool. We showed an application example using the developed tool.

In Chap. 6, we give the conclusion and future work of the development method in the software evolution.

Chapter 2

Preliminary

First, we give the definition of Software Evolution, Petri net, workflow net and process tree.

2.1 Software Evolution

Software evolution [30] is a process of developing software and updating it repetitively for various reasons such as improving the performance and security of the software. In some cases, the process involves applying patches or security fixes to the software. During the software development life-cycle, program code changes repetitively to adapt to new requirements and improvements.

In this paper, we focus on the behavioral changes in software throughout its evolution. During the implementation of the software specifications, upgrades or patches are applied to its source code. Changes of software's behavior are expected after the modifications. In other words, the behavior of the software also evolves during the modification process. In this case, behavioral analysis needs to be done repetitively at every development cycle.

2.2 Petri Net

Petri nets [31, 32, 33] are a mathematical and graphical modeling tool applicable to many systems. They are promising tool for describing and studying information processing systems that are characterizes as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic. We give the definition of petri net as follows:

Definition 2.1 *Petri net is a three tuple* N=(P,T,A)*, where* P*,* T*, and* A $(\subseteq(P\times T)\cup(T\times P))$ *are finite sets of places, transitions, and arcs, respectively.* □

Let x be a node of N. $\overset{N}{\bullet}x$ and $x\overset{N}{\bullet}$ respectively denote $\{y|(y,x)\in A\}$ and $\{y|(x,y)\in A\}$. A marking (or a state) is a mapping M: $P\rightarrow\mathbb{N}$. We represent M as a bag over P: $M=[p^{M(p)}|p\in P, M(p)>0]$. A transition t is said to be firable in M if $M\geq\overset{N}{\bullet}t$. Firing t in M results in a new marking M' ($=M\cup t\overset{N}{\bullet}-\overset{N}{\bullet}t$). This is denoted by $M[N,t\rangle M'$. A marking M_n is said to be reachable from a marking M_0 if there exists a transition sequence $t_1t_2\cdots t_n$ such that $M_0[N,t_1\rangle M_1[N,t_2\rangle M_2\cdots[N,t_n\rangle M_n$. The set of all markings reachable from M_0 in (N,M_0) is denoted by $R(N,M_0)$. The tree representation of the markings in $R(N,M_0)$ is called the reachability tree.

2.3 Workflow Net

Petri net can model workflow procedures and have typical properties. First of all they always have two special places; the source place p_I and the sink place p_O. The places corresponds to the beginning and termination of the processing of a case respectively. Secondly, for each transition t (place p) there should be directed path from the source place to the sink place. A Petri net which satisfies this requirements is called as a Workflow Net (WF-net) [34]. Figure 2.1 illustrate the structure of a WF-net. WF-net can be defined as follows:

Definition 2.2 *N is said to be a WF-net if (i) N has a single source place p_I and a single sink place p_O; and (ii) every node is on a path from p_I to p_O; and (iii) there is no dead transition in N.* □

Figure 2.1: The structure of WF-net.

There is a particular subclass of WF-nets: *Well-Structured* (WS for short). A structural characterization of good workflows is that two paths initiated by a transition (a place) should not be joined by a place (a transition). WS is derived from this structural characterization. To give the formal definition of WS, we introduce some notations. We make N strongly connected by connecting p_O to p_I via an additional transition t^*. The resulting Petri net is called the *short-circuited net* of N, and is denoted by $\overline{N}$ ($=(P, T\cup\{t^*\}, A\cup\{(p_O,t^*),(t^*,p_I)\}\})$). Let c be a circuit in $\overline{N}$. A path $\rho = x_1x_2\cdots x_n$ ($n\geq 2$) is called a *handle* [53] of c if ρ shares exactly two nodes, x_1 and x_n, with c.

A path ρ is called a bridge between c and its handle h if each of c and h shares exactly one node, x_1 or x_n, with ρ. A path $b = x_1x_2\cdots x_n$ ($n\geq 2$) is a *bridge* between c and a handle of c if each of h and c shares exactly one node, x_1 or x_n, with b.

A handle (a bridge) from a node x to another node y is denoted by a XY-handle (a XY-bridge), where if $x\in P$ then X is P, otherwise X is T; if $y\in P$ then Y is P, otherwise Y is T. A handle (a bridge) from a place to another place is called a PP-handle (a PP-bridge). A handle (a bridge) from a place to a transition is called a PT-handle (a PT-bridge). A handle (a bridge) from a transition to a place is called a TP-handle (a TP-bridge). A handle (a bridge) from a transition to another transition is called a TT-handle (a TT-bridge). A WF-net N is said to be WS if there are neither TP-handles nor PT-handles of any circuit in $\overline{N}$. We can decide in polynomial time whether a given WF-net is WS by applying a modified version of the max-flow min-cut technique [34].

Adding some constraints to the graph structure of WF-nets, we can obtain the subclasses of WF-nets. There are the following major subclasses:

(i) State Machine (SM) WF-net: A WF-net is State Machine iff each transition has exactly one input place and one output place ($\forall t$, $|\bullet t|=|t\bullet|=1$).

(ii) Marked Graph（ MG）WF-net: A WF-net is Marked Graph iff each transition has exactly one input transition and one output transition ($\forall p$, $|\bullet p|=|p\bullet|=1$).

(iii) Well-Structured (WS) WF-net: A WF-net is Well-Structured iff the short-circuited net has neither TP-handle nor PT-handle.

(iv) Free Choice（ FC）WF-net: A WF-net is Free Choice iff each transition has exactly one input transition and one output transition ($\forall p_1, p_2$, $p_1\bullet\cap p_2\bullet\neq\emptyset \Rightarrow |p_1\bullet|=|p_2\bullet|=1$). and one output transition ($\forall p_1, p_2$, $p_1\bullet\cap p_2\bullet\neq\emptyset \Rightarrow p_1\bullet=p_2\bullet$).

2.4 ST-net

Van Hee et al. [38] proved that WF-nets which is constructed by state machines and cycle-free marked graph is sound, serialisable and split-separable. We call this nets as ST-nets. These nets are constructed by means of refinement. In many case, modeling problems can be solved by (provably correct) ST-nets. The definition is given in Def. 2.3.

Definition 2.3 *The set N of ST-nets is the smallest set of nets N defined as follows: - if N is an acyclic MG WF-net, then N* $\in \mathcal{N}$.

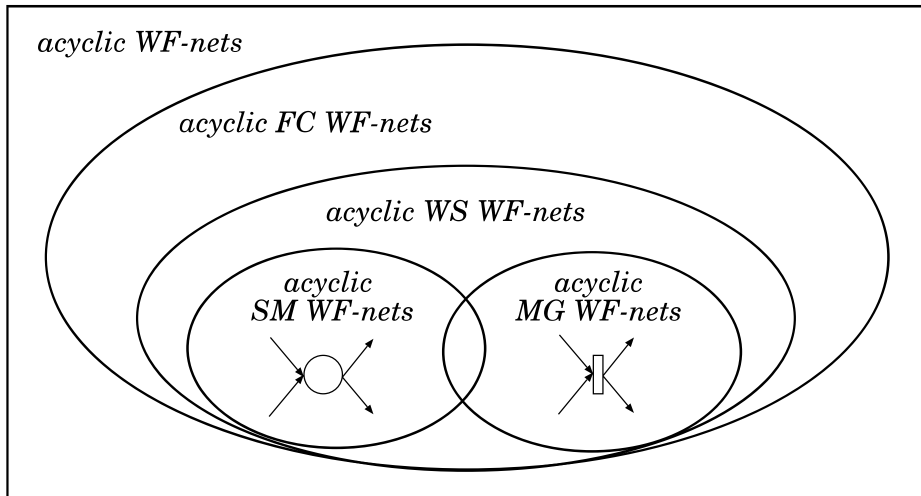


Figure 2.2: Petri net classes.

- if N is an SM WF-net, then $N \in \mathcal{N}$.
- if $N \in \mathcal{N}$, $s \in S_N$ *and* $M \in N$ *is an sWF-net, then* $N \otimes_s M \mathcal{N}$.
- if $N \in \mathcal{N}$, $t \in T_N$ *and* $M \in N$ *is an tWF-net, then* $N \otimes_t M \mathcal{N}$.

2.5 Soundness

Soundness [34] is a criterion of correctness for workflow definitions.

Definition 2.4 *A WF-net N (=*(P, T, A, ℓ)*) is said to be sound iff*
(i) $\forall M \in R(N, [p_I])$: $\exists M' \in R(N, M)$: $M' \geq [p_O]$*; and*
(ii) $\forall M \in R(N, [p_I])$: $M \geq [p_O] \Rightarrow M = [p_O]$*; and*
(iii) There is no dead transition in $(N, [p_I])$.
A WF-net N is sound iff $(\overline{N}, [p_I])$ *is live and bounded.* □

2.6 Process Tree

A process tree is a tree representation of a process [**?**]. Each leaf node and each internal node respectively represent an action and an operator in the process. In this paper, we use four operators standardized by The Workflow Management Coalition (WfMC for short) [34] : sequence ($\rightarrow$), exclusive-choice ($\times$), parallel ($\wedge$) and loop ($\circlearrowleft$).

Definition 2.5 *The set* Π *of process trees* π *is as follows:*

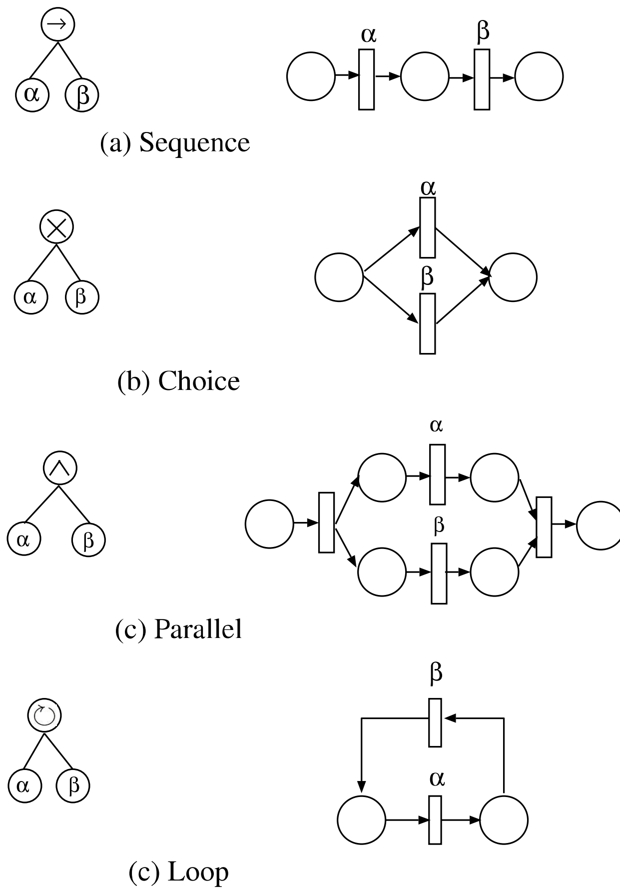


Figure 2.3: Translation of process tree operators to Petri net constructs

(i) If ℓ is an action label, then $\ell \in \Pi$.

(ii) If $\oplus$ is an operator and $\ell_1, \ell_2, \cdots, \ell_n$ are action labels, then $\oplus(\ell_1, \ell_2, \cdots, \ell_n) \in \Pi$.

(iii) If $\oplus$ is an operator and $\pi_1, \pi_2, \cdots, \pi_n \in \Pi$, then $\oplus(\pi_1, \pi_2, \cdots, \pi_n) \in \Pi$. □

Each operator can be translated to a part of a WF-net (see Fig. 2.3) and its equivalent process tree formula $\oplus(\pi_1, \pi_2, \cdots, \pi_n)$.

In the process tree, the position for each leaf node in the process tree is assigned with depth-first search such that $t^{(i)}(i{=}1, 2, \cdots, n)$. It is shown from left to right in ascending order of each node. For $t^{(i)}$ and $t^{(j)}$ in a process tree Π, the position of $t^{(i)}$ is on the left of $t^{(j)}$ if $i < j$ holds.

2.7 Behavioral Inheritance

Behavioral inheritance is a relaxation of branching bisimilarity [39, 40], which is an equivalence relation on WF-nets. Branching bisimilarity allows some transitions not to be observed. Such transitions are denoted by a designated label τ. Intuitively, branching bisimilarity equates WF-nets whose observable behaviors are the same.

Definition 2.6 (Branching Bisimilarity) Let G_{N_X} and G_{N_Y} be respectively the reachability graphs of a WF-net $(N_X, [p_I^X])$ and another WF-net $(N_Y, [p_I^Y])$. A binary relation $\mathcal{R}$ $(\subseteq R(N_X, [p_I^X]) \times R(N_Y, [p_I^Y]))$ is branching bisimulation iff

(i) If $M_X \mathcal{R} M_Y$ and $M_X[N_X, \alpha\rangle M_X'$, then $\exists M_Y', M_Y'' \in R(N_Y, [p_I^Y]): M_Y[N_Y, \tau^*\rangle M_Y''$, $M_Y''[N_Y, (\alpha)\rangle M_Y'$, $M_X \mathcal{R} M_Y''$, and $M_X' \mathcal{R} M_Y'$;

(ii) If $M_X \mathcal{R} M_Y$ and $M_Y[N_Y, \alpha\rangle M_Y'$, then $\exists M_X', M_X'' \in R(N_X, [p_I^X]): M_X[N_X, \tau^*\rangle M_X''$, $M_X''[N_X, (\alpha)\rangle M_X'$, $M_X'' \mathcal{R} M_Y$, $M_X' \mathcal{R} M_Y'$; and

(iii) If $M_X \mathcal{R} M_Y$ then $(M_X = [p_O^X] \Rightarrow M_Y[N_Y, \tau^*\rangle[p_O^Y])$ and $(M_Y = [p_O^Y] \Rightarrow M_X[N_X, \tau^*\rangle[p_O^X])$.

$(N_X, [p_I^X])$ and $(N_Y, [p_I^Y])$ are said to be branching bisimilar, denoted by $(N_X, [p_I^X]) \sim_b (N_Y, [p_I^Y])$, iff there exists a branching bisimulation $\mathcal{R}$ between G_{N_X} and G_{N_Y}. □

See the illustration of branching bisimilarity in Fig. 2.4. There exists the same observable marking in both G_{N_X} and G_{N_Y}.

Definition 2.7 (Life-cycle Inheritance) To give the formal definition of life-cycle inheritance [34], we use two operators: (i) *encapsulation* and (ii) *abstraction*. (i) For any $H(\subseteq \mathcal{A})$, the encapsulation operator ∂_H is a function that removes all transitions with a label in H from N. Formally, $\partial_H : N \mapsto (P, T', A', \ell')$ such that $T' = \{t \in T \mid \ell(t) \notin H\}$, $A' = A \cap ((P \times T') \cup (T' \times P))$, and $\ell' = \ell \cap (T' \times (\mathcal{A} \cup \{\tau\})$. (ii) For any $I(\subseteq \mathcal{A})$, the abstraction operator τ_I renames all transition labels in I to τ. Formally, $\tau_I : N \mapsto (P, T, A, \ell')$ and for any $t \in T$, $\ell(t) \in I$ implies $\ell'(t) = \tau$ and $\ell(t) \notin I$ implies $\ell'(t) = \ell(t)$. For any $\ell(\subseteq \mathcal{A})$, using that operator, we give the definition of life-cycle inheritance: N_X is a subclass under life-cycle inheritance of N_Y iff there is an $I(\subseteq \mathcal{A})$ and an $H(\subseteq \mathcal{A})$ such that $I \cap H = \phi$ and $\tau_I \circ (\partial_H(N_X), [p_I^X]) \sim_b (N_Y, [p_I^Y])$. If N_Y is a subclass of N_X under life-cycle inheritance, then N_Y is said to inherits the behavior of N_X.

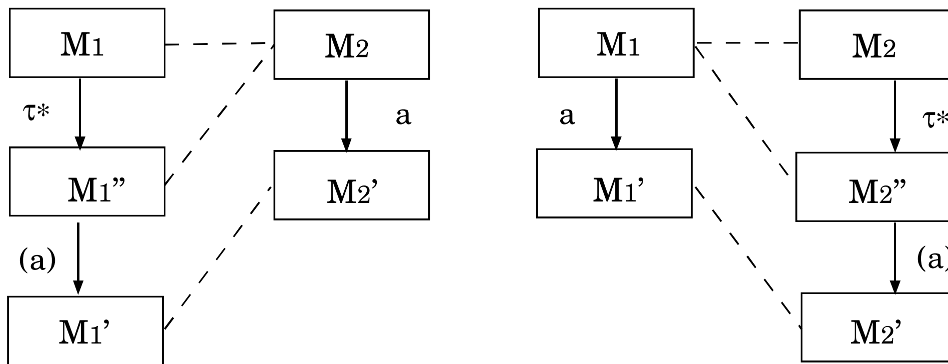


Figure 2.4: Illustration of branching bisimilarity. There exists the same observable marking in both G_{N_X} (left) and G_{N_Y} (right).

2.8 Backward Compatibility

A software that can work with the previous version is called as backward compatible. Backward compatibility is an important property to ensure that newer version of software can execute the task or function that exist in the previous version.

We can define backward compatibility as follows:

Definition 2.8 *A software Y is said to be backward compatible to the older version software X if software Y has the functions that exist in software X.* □

Definition 2.8 shows the intuitive definition of backward compatibility for software. In terms of WF-net, backward compatibility is related to behavioral inheritance. A backward compatible software inherits the behavior of the previous version. Backward compatibility can be verified with behavioral inheritance, however, checking the behavior of program is intractable for large and complex programs.

Figure 2.5, shows an example of backward compatibility concept. For a software version *X* which has *Function A* and *Function B*. Then once upgrade is applied on *X*, we obtain version *Y*. Software version *Y* has *Function A*, *Function B* and new *Function C*. Version *Y* preserved both *Function A* and *Function B*, therefore *Y* is backward compatible. As another example, we upgrade version *Y* to newer version *Z*. Version *Z* has *Function A*, *Function C* and new *Function D*. However, version *Z* do not preserve *Function B*. We can conclude that version *Z* is not backward compatible.

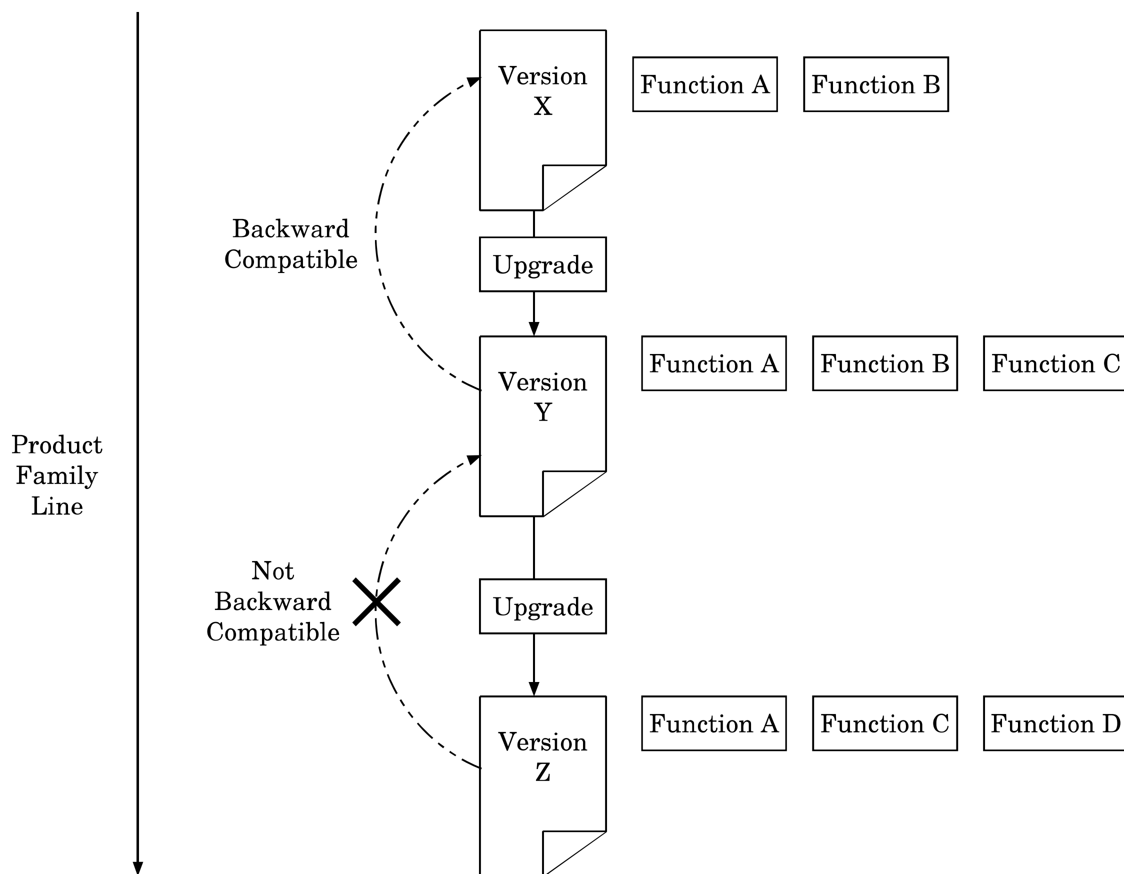


Figure 2.5: Illustration of backward compatibility.

2.9 UML Activity Diagram

Activity diagram [5] is an important diagram to describe dynamic aspect in UML. It can be used to express execution sequence, concurrency and condition branching in a form of activity flow chart. Activity diagram is important to capture control and object flow of a process that is very useful in software development.

Chapter 3

Software Evolution and Petri Net Approach

In software development, to follow product line, we must preserve certain important functions. However, due to coding mistakes or omission of change, the development miss can lead to compatibility problem. However, it is hard to verify only by looking at the source code or execute the program through the debug process.

We first need to formalize the problem in software evolution. As new software is developed, it is harder to verify the compatibility of the software due to its increasing complexity. We propose our approach based on Petri net model. Our approach is model-based, therefore preservation of functions can be guaranteed. In this chapter, after the problem introduction, we show our approach and show how we can analyze program with our model-based techniques.

3.1 Real World Concept of Software Evolution

In this section, we discuss real world problem of software evolution [41, 29]. We know that during the evolution of software, it is important to manage and preserve the product line of the software. As software evolves to adapt to requirements of consumer over time, the newer version of the software tends to become more complex.

Figure 3.1 shows the relation of software development and product family line . Incremental and iterative software development process can produces new releases. The new releases of software requires new version such as 1.0, 1.1, 1.2 and 1.3. Version 1.3 shows the software is applied with minor upgrade. Version 1.0, 2.0 and 3.0 show that the software are applied with major upgrade. The series of version numbering represents the product family line of the software. Minor upgrade is an upgrade that changes the behavior of only some part of the software

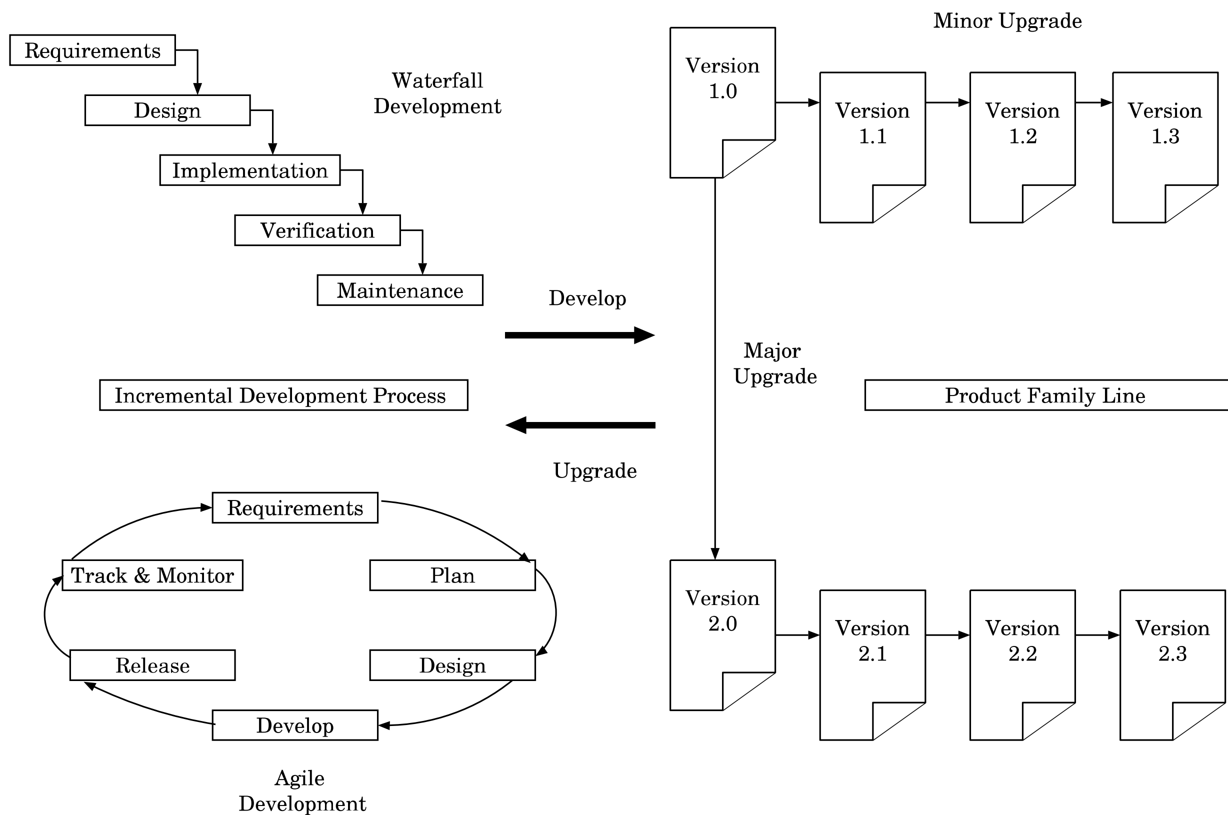


Figure 3.1: Product family line in software evolution.

such as in some extension or functions. Major upgrade is an upgrade that changes the behavior of the software on its core functions such as its performance or main functionality in the software. Well-known development process such as waterfall model and agile model only focus on the process to produce new version. Our research aims to preserve the backward compatibility in the product family line.

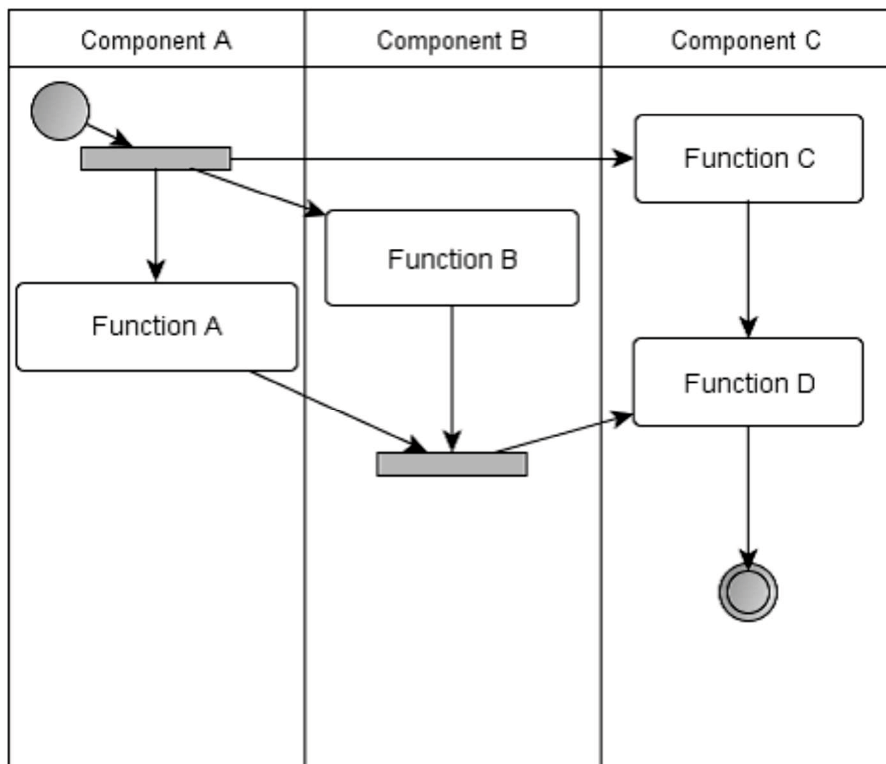
Software evolves as it goes through the development process and undergo new releases. The development and upgrade process assign version number to the software to show that the software was applied with minor or major upgrade. It is important to follow a given specifications in order to maintain the product family line.

3.1.1 Derivative Development

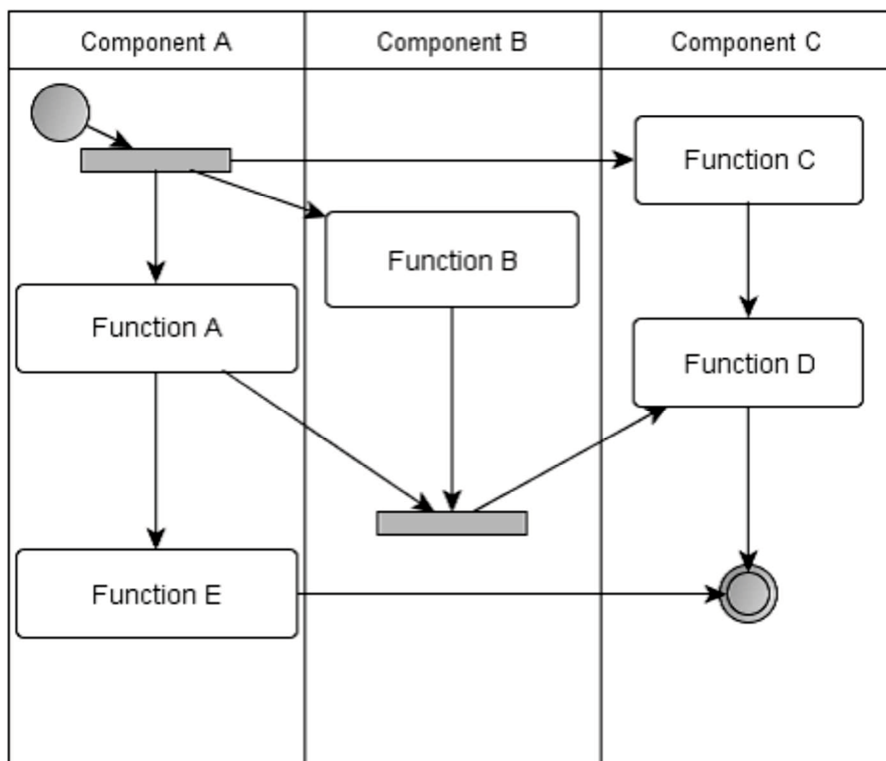
Derivative development [2, 3] involves extension of product design and verification of backward compatibility. Generally, developers are restricted only to verification of logical correctness. Product design such as UML activity diagram only focus on attributes but not on its structure. Therefore, backward compatibility verification is not well-considered in conventional design

method such as with UML.

Let us take Fig. 3.2 as a simple example. We have an original product which consists of three components with related functions between those components. In Fig. 3.2(a) we have functions that we call as Function A, B, C and D. Function A, B and C is concurrently executed and Function D is executed after Function C before terminating. All order in these functions are very important as part of the component's functionality. The order of the execution cannot be changed. However, during new product development, a new function needs to be extended. Figure 3.2(b) shows the extended product which was added with the execution of Function E after the parallel execution of Function A and B. The extended product is now called the derivation of the original product.



(a) Original model



(b) Extended model

Figure 3.2: Example of UML activity diagrams.

3.1.2 Real World Example

We give an example of products' derivatives of an smart refrigerator (i-Refrigerator for short) [43]. Figure 3.3 shows the concept of the refrigerator. Figure 3.4 shows the activity diagram that describes the concept of i-Refrigerator before extension. The diagram shows a flow starting with the user inserting food products into the i-Refrigerator, then concurrently sends the product data and its expiry date through RFID reader to the touch panel and database. Let us say there is a requirement to measure the precise weight of each product such as for using the product as cooking recipe. So we add an electronic balance that measures and relays products' weight data to the touch panel and the database. Therefore, we need to verify that the extended i-Refrigerator is a valid derivation of the original product in Fig. 3.3. Figure 3.25 shows program π of the i-Refrigerator. Figure 3.6 of a C program π_2 for i-Refrigerator *updateDB()* function.

Concretely, let us say that a new version 2.0 was developed by the manufacturer which upgraded the input to barcode reader to make input easier. We can say that version 2.0 is an upgraded version of version 1.0. Finally, in version 3.0 the manufacturer developed a more advance input which use RFID. Let us say that the RFID input has satisfied the demand and to make the i-Refrigerator more convenience, the manufacturer added an electronic balance to measure weight of the food products. By extending a weight measurement functionality to version 3.0 that produce version 3.1, we can say that version 3.1 is a derivative of version 3.0. This requirement only extends the function and do not change other existing function in the i-Refrigerator. Thus, we can say the changes is a minor upgrade.

Next, let us say there are requirement to improve the performance of the database. We need to upgrade function *updateDB()* in program π_2. We can change the behavior of program π_2 by parallelizing the update process. Thus, we can say that it is a major upgrade for the database function.

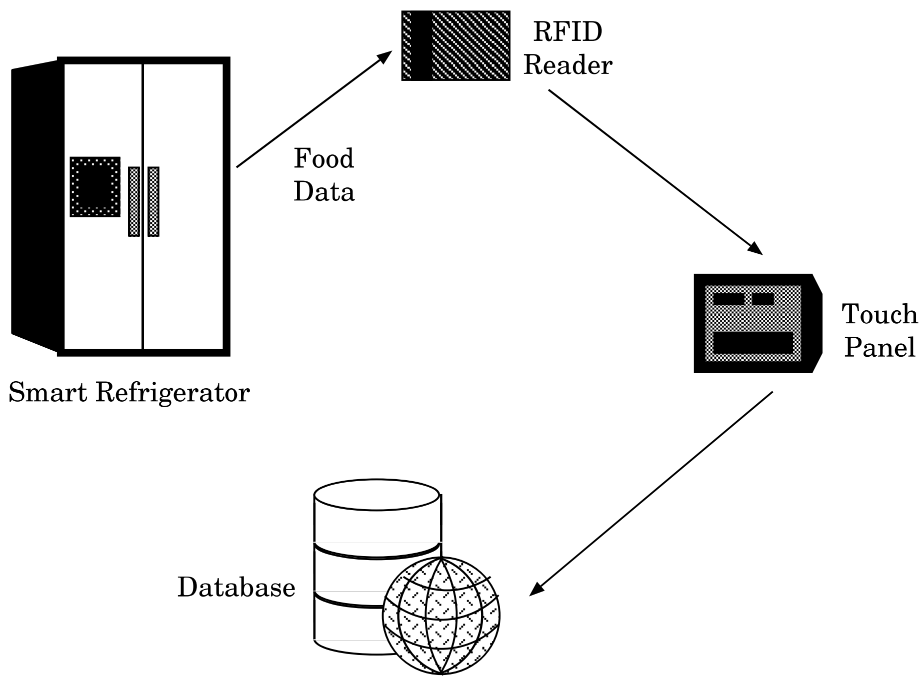


Figure 3.3: An i-Refrigerator design.

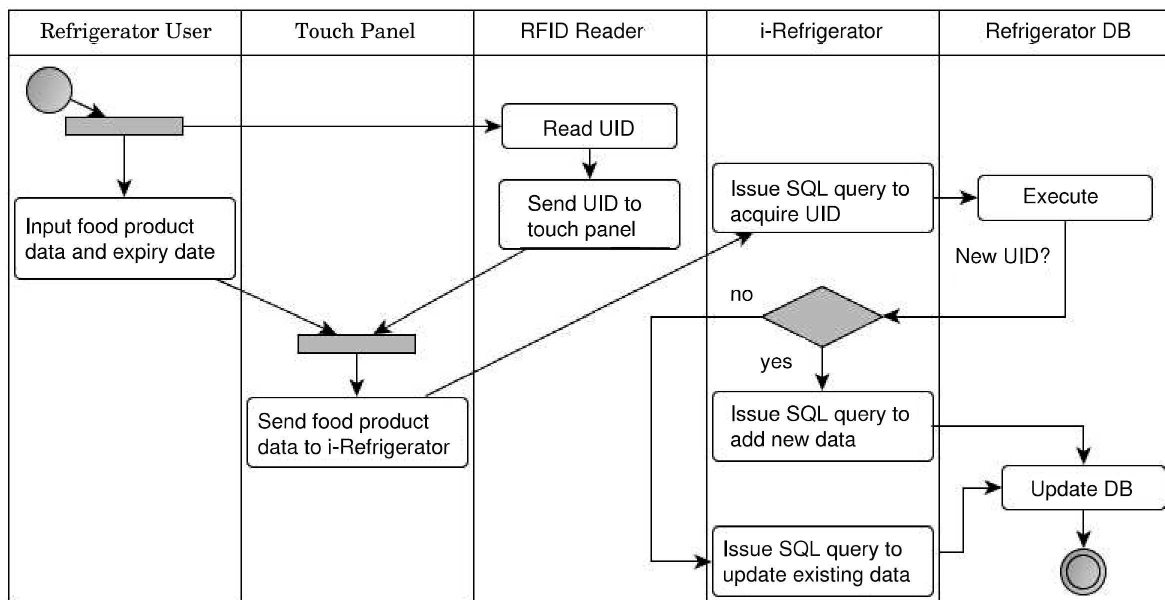


Figure 3.4: i-Refrigerator original design.

```
 1 void main(int argc, char* argv[]) {
 2 serialNum = ReadUID();
 3 RFIDSensorCheck();
 4 food=InputData(foodName, producer, serialNum, expDate);
 5 if((pid = fork()) != 0){
 6   initDB();
 7   int UID = IssueSQLQuery(serialNum, food);
 8   if(!existingInDB(UID)){
 9    IssueNewData(UID);
10     StoreUID(UID, food);
11   }
12  updateDB();
13 }else {
14 usleep(10000);
15 RFIDSensorCheck();
16    serialNum=ReadUID();
17 if(checkExpiryDate(serialNum);){
18      sendExpiredDateWarning();
19 }
20 }
21 if(pid == 0)
22    _exit(status);
23 else
24    while(kill(pid, 0));
25     wait(&status);
26 StandbyMode();
27 updateDB();
28 }
```

Figure 3.5: A C program π for i-Refrigerator.

```
 1 void updateDB(){
 2  int stock = getStockNumber();
 3  for(i=1;i<=stock;i++)
 4  {
 5   if(i<=(stock/2)){
 6    if(i%2==0){
 7     setID("ID:XA-00%d\n",i);
 8     type = meat';
 9    }else{
10     setID"ID:XB-00%d\n",i);
11        type = packed-food';
12     }
13   }else if(i>(stock/2)+1)
14   {
15     setID("ID:XC-00%d \n",i);
16     type = vegetables';
17   }
18
19   if(type=='A'){
20      storeProduct("XA-00",i);
21   }else if(type=='B'){
22      storeProduct("XB-00",i);
23   }else if(type=='C'){
24      storeProduct("XC-00",i);
25   }
26 }
```

Figure 3.6: A C program π_2 for i-Refrigerator *updateDB()* function.

3.2 Problem in Software Evolution

We can analyze the behavior of the model using Petri net analysis technique. We know that a program can be translated into WF-net model. Hence, we can convert the problem of protocol implementation verification into behavioral analysis of WF-net. Concretely, we can reduce the real world verification problem in Fig. 3.7 to the WF-net verification problem.

3.2.1 Backward Compatibility in Software Evolution

Intuitively, we can say that for a specification α, a program π implements α if π can do what α do. If so, we call π as an implementation of α [42]. All the tasks in π present in α, but in general, π would add new tasks. Even if so, π must be still an implementation of α. Hence, in terms of WF-net, we give the following definition:

Definition 3.1
For a program X and a protocol specification α, X is said to implements α iff N_X is a subclass of N_α. □

We illustrate Def. 1 in Fig. 3.8(left). To explain this situation, we say that π inherits the protocol of α if the WF-net N_X representing program π has the same behavior as N_α after blocking the execution of any added tasks. We need only to check the behavioral inheritance between N_X and N_α.

In the software evolution perspective, the problem is reduced to backward compatibility problem. In general, N_X can evolves into a newer version N_Y. We can always verify the behavior of N_α in N_Y directly but by ignoring the condition that N_Y is a subclass of N_X. In some cases, although N_Y inherits the behavior of N_α, N_Y does not always inherits the behavior of N_X. In fact, N_Y that evolves from its superclass N_X can also evolves into newer versions N_Y', N_Y'', N_Y''' and so on. In our approach, by validating the backward compatibility of N_Y with the previous version N_X, we can ensure that N_X is the superclass of N_Y and its newer version in the software production line. We give the following definition on the backward compatibility of N_Y to N_X.

Definition 3.2
For a program Y and its previous version X, Y is said to be backward compatible with X iff N_Y is a subclass of N_X. □

From Def. 4, we can say that N_Y is backward compatible with N_X if N_X is the superclass of N_Y. We illustrate Def. 4 in Fig. 3.8(right). We can verify the behavioral inheritance between

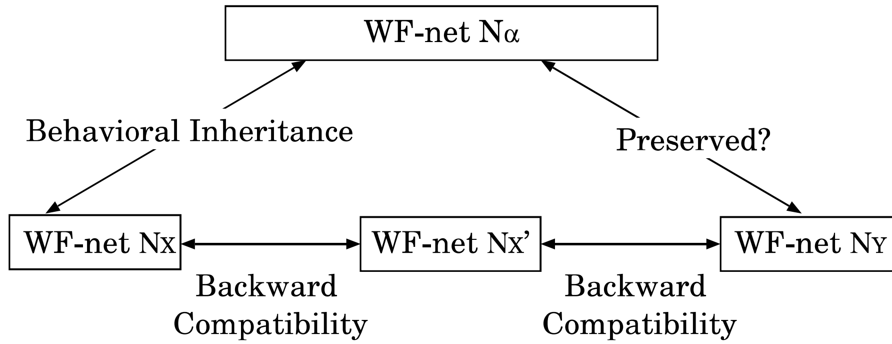


Figure 3.7: Illustration of our verification problem.

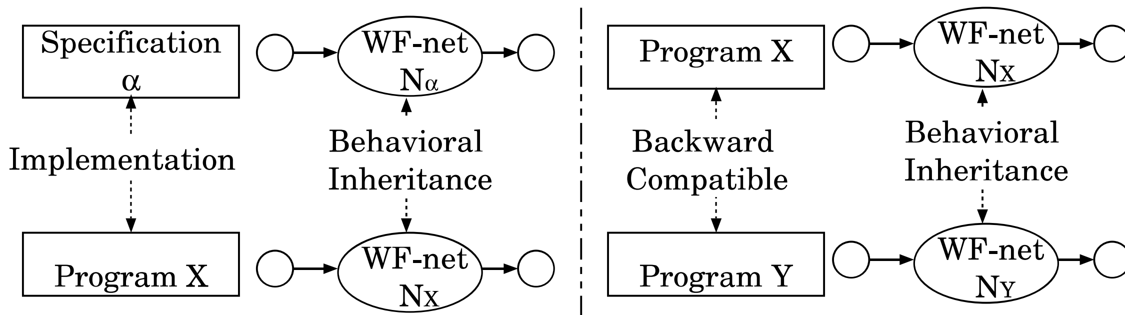


Figure 3.8: Relation of implementation and backward compatibility with behavioral inheritance.

N_α and N_Y by verifying the backward compatibility of N_Y with N_X first. Here, we introduced a notion of behavioral inheritance called as life-cycle inheritance (See Appendix B). From Def. 4 and Def. 5, we deduce the following theorem:

Theorem 3.1
For a program X, its newer version Y and a specification α, Y implements α iff WF-net N_Y is a subclass of WF-net N_X where N_X is a subclass of N_α under life-cycle inheritance. □

Proof: We first prove the "if" part. Life-cycle inheritance relation is transitive. If N_α is the superclass of a WF-net N_X then its subclass N_Y and the subclasses of its subclass N'_Y, N''_Y, N'''_Y is also the subclass of N_α under life-cycle inheritance.

Next, we prove the "only if" part. From Def. 4, if N_X is not the subclass of N_α under life-cycle inheritance then Y does not implement α. From Def. 5, if N_Y is not a subclass of N_X under life-cycle inheritance, Y is not backward compatible with X. If Y is not backward compatible with X, then Y does not implement α. **Q.E.D.**

Theorem 3.1 implies the transitive relation of behavioral inheritance in software evolution. If N_Y inherits the behavior of N_α, then protocol α is implemented in program Y. From the proof,

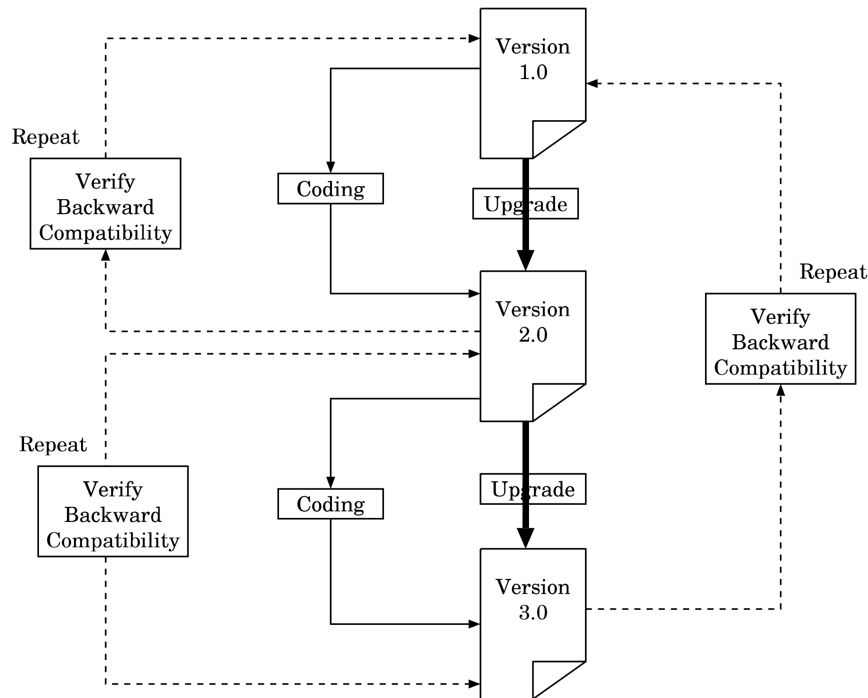


Figure 3.9: Conventional approach.

we only need to check the relation shown in Fig. 3.7 and Fig. 3.8. If a WF-net N_X of program X inherits the behavior of N_α then program X implements α. Then let us consider where a new version Y derived from X. If N_Y inherits the behavior of N_X then N_Y also inherits the behavior of N_α. Thus we can say that Y also implements α.

3.2.2 Backward Compatibility Verification Problem

Backward compatibility should satisfies (i) behavioral inheritance and (ii) response property. This two conditions can be used to verify backward compatibility when minor or major upgrades are applied. Minor upgrades only changes some part of the software component such as minor functions, classes or interface in the software. On the other hand, major upgrades changes the core functionality of the whole program in the software such as performance, processing sequence or most structure of the program.

In conventional method, backward compatibility is always being verified as shown in Fig. 3.9. It is a repetitive process at each development stage where developer has to verify each derivatives once upgrade is applied. Our approach reduce the repetitive process by preserving the backward compatibility at each development stage as shown in Fig. 3.10.

Based on the condition of behavioral inheritance and response property, we can verify back-

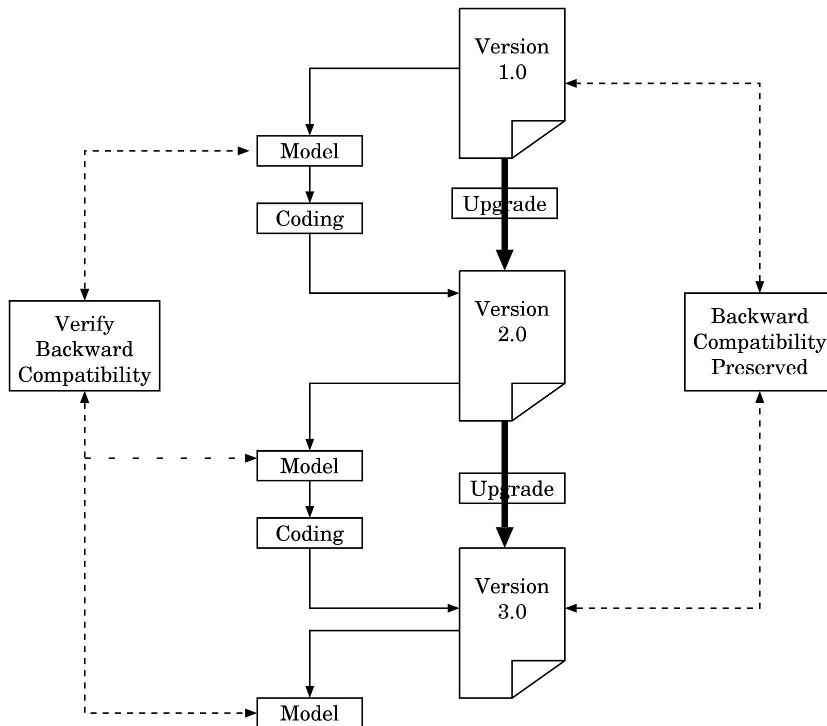


Figure 3.10: Our approach.

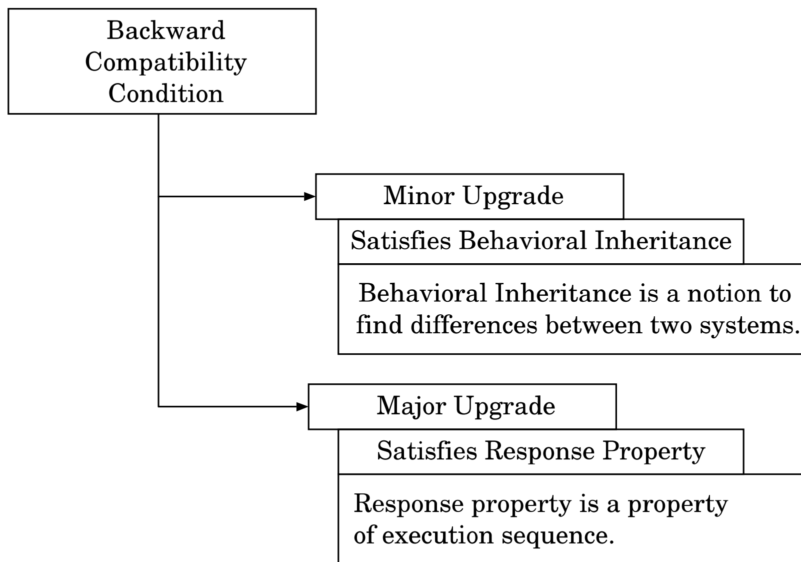


Figure 3.11: Conditions of backward compatibility.

ward compatibility. We summarize the conditions of backward compatibility as shown in Fig. 3.11.

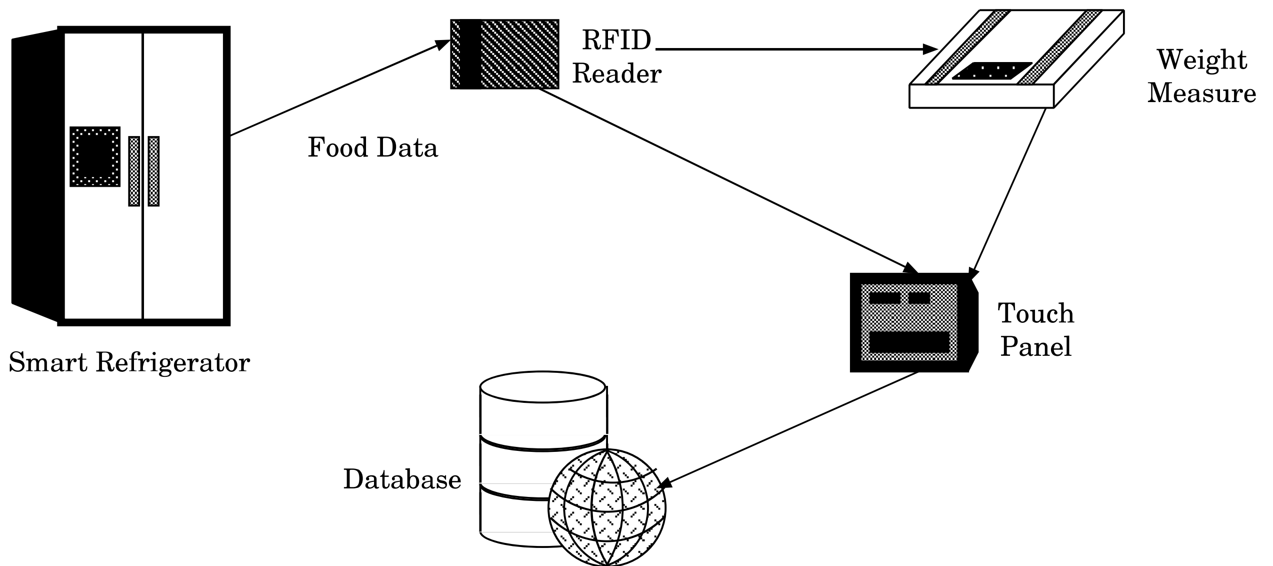


Figure 3.12: An extended i-Refrigerator design.

Problem 1 : Minor Upgrade

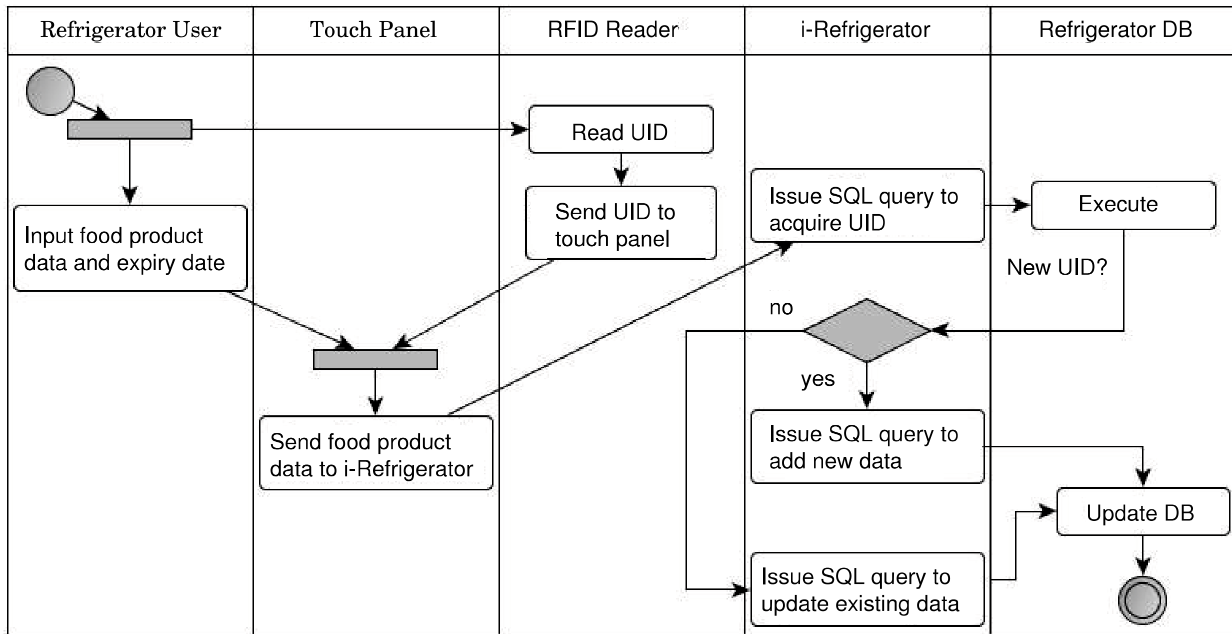
Behavioral inheritance is an important property to verify the behavior of the software when minor upgrades are applied to the software. We can identify the differences of behavior in some part of the program. Therefore, software which satisfies behavioral inheritance preserves the behavior of its previous version. This can only be verified for small changes in certain function of classes such as minor upgrades because it does not alter the core functions of the software.

We take an example for the design of i-Refrigerator to be extended as shown in Fig. 3.12. The UML activity diagram is shown in Fig. 3.2.2 and Fig. 3.13. Let us say there is a requirement to upgrade the original design model in Fig. 3.2.2 by adding the *Measure Weight* function as shown in Fig. 3.13. C program π_3 shows the new i-Refrigerator extended with measure weight.

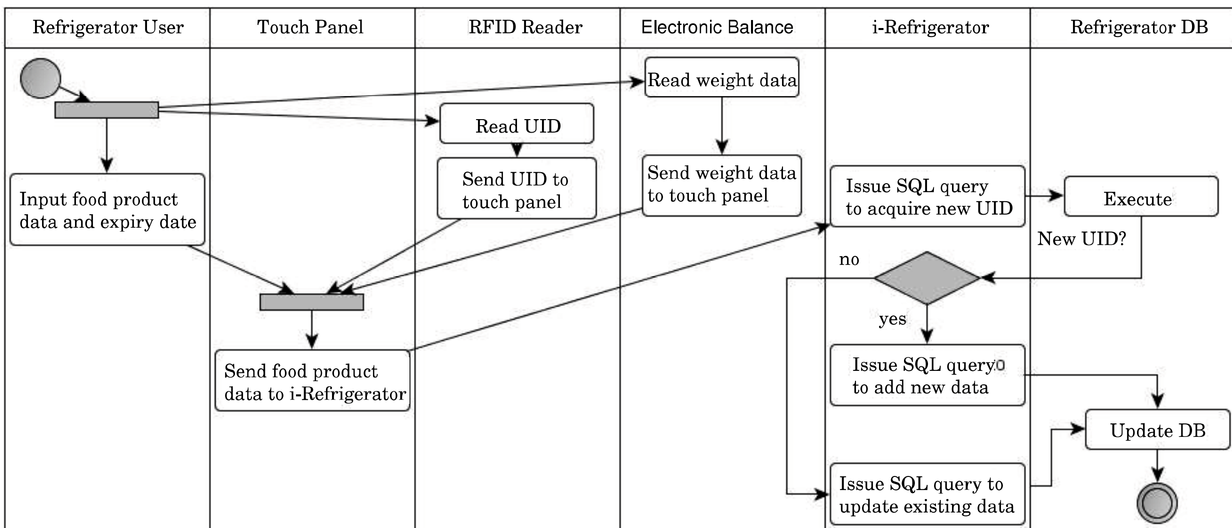
In this minor upgrade, we need to analyze whether program π_3 is backward compatible with program π or not.

Problem 2 : Major Upgrade

Response property is a property for execution sequence. Response property is useful to analyze the execution sequence within a program for major upgrades. For example, in a program π_α shown in Fig. 3.15 where statement A, B, C, D and H exists, let us say we upgraded the program into a new program π_β as shown in Fig. 3.16. In this case we, need to confirm whether A executes before B and D executes in after A or not. This execution sequence is very important to verify



(a) Original model



(b) Extended model

Figure 3.13: UML activity diagrams of i-Refrigerator.

if there are major changes in the program. Difference execution sequence clearly have different behavior in the program. Response property is also related to behavioral inheritance but is limited a specifications that we need to verify in the program.

```
 1 void main(int argc, char* argv[]) {
 2 serialNum = ReadUID();
 3 RFIDSensorCheck();
 4 weight = WeightMeasure();
 5 if(validateWeight(serialNum, food){
 6  if(weight > 10000){
 7  sendOverweightMessage();
 8  }else{
 9   food=InputData(foodName, producer, serialNum, expDate, weight);
10 }
11 if((pid = fork()) != 0){
12   initDB();
13   int UID = IssueSQLQuery(serialNum, food);
14   if(!existingInDB(UID)){
15    IssueNewData(UID);
16     StoreUID(UID, food);
17   }
18  updateDB();
19 }else {
20 usleep(10000);
21 RFIDSensorCheck();
22    serialNum=ReadUID();
23 if(checkExpiryDate(serialNum);){
24      sendExpiredDateWarning();
25 }
26 }
27 if(pid == 0)
28    _exit(status);
29 else
30    while(kill(pid, 0));
31     wait(&status);
32 StandbyMode();
33 updateDB();
34 }
```

Figure 3.14: A C program π_3 for new i-Refrigerator extended with measure weight.

```
 1 void main(int argc, char* argv[]) {
 2 int status, pid = -2;
 3 int delete = 0, save = 0;
 4 A
 5 B
 6 if((pid = fork()) != 0){
 7      if(delete == 1){
 8          C
 9       }
10          D
11  } if(pid == 0)
12     _exit(status);
13  else
14     while(kill(pid, 0));
15      wait(&status);
16      H
17 }
```

Figure 3.15: A C program π_α.

```
 1 void main(int argc, char* argv[]) {
 2 int status, pid = -2;
 3 int delete = 0, save = 0;
 4 A
 5 B
 6 if((pid = fork()) != 0){
 7      if(delete == 1){
 8          C
 9      }else{
10         D
11         E
12       }
13 }else {
14    G
15 }
16 if(pid == 0)
17    _exit(status);
18 else
19    while(kill(pid, 0));
20     wait(&status);
21     H
22 }
```

Figure 3.16: A C program π_β.

3.3 Our Approach

We propose two major methods in our validation approach; (i) translation of program to Petri net model, and (ii) model-based security protocol verification. We reverse engineer a program source code by translating it into a Petri net model. Then, we propose an approach to verify security protocol implementation inside the source code. See Fig. 3.17.

In the first method, we propose our translation method with a tool called as C2PNML. A C program can be reversed engineered to extract software model from the source code. The software model is represented as Petri net model N_A. Then we can implement a specification into the program. Concretely, specification is represented as Petri net model N_B. We assume that the security protocol in the program has already been implemented. Thus we can obtain Petri net model N_C from the program implemented with security protocol. So we need to verify if model N_C inherits the behavior of model N_B or not after the implementation. Next, we can check whether Petri net model N_B is implemented into the Petri net model N_C by analyzing the behavioral inheritance of the program. Finally, we can obtain the result whether model N_C implements N_B or not.

3.4 Translation of Program Structure into Petri Net

In this section, we propose a reverse engineering method to translate a program back to its design model so we can use it to verify its behavior. Generally, a program is constructed based on its design model. However, the structure of the program will always change. Most compilers are made to identify grammatical and syntax errors before compilation. Although the program passes the compilation, it depends on the programmer to verify the behavior of the program.

There are some programs proposed to convert source code into flow chart such as Code2Flow [44], Code Visual to Flowchart [45] and AutoFlowChart [44]. However, converting to flowchart only help the developer to visualize the whole program structure. Therefore, we need not only the visualization of the code but the behavioral analysis of the program itself. Existing tools also cannot visualize parallel programs.

We propose a translation approach to convert parallel structured program into Petri net model. Once a program is converted into Petri net model, we can analyze it with Petri net technique. There are a lot of behavioral analysis techniques that can be applied to the Petri net model. Our overall approach for the translation is shown in Fig. 3.18.

In this section, we propose a tool called as C2PNML to convert program source code into

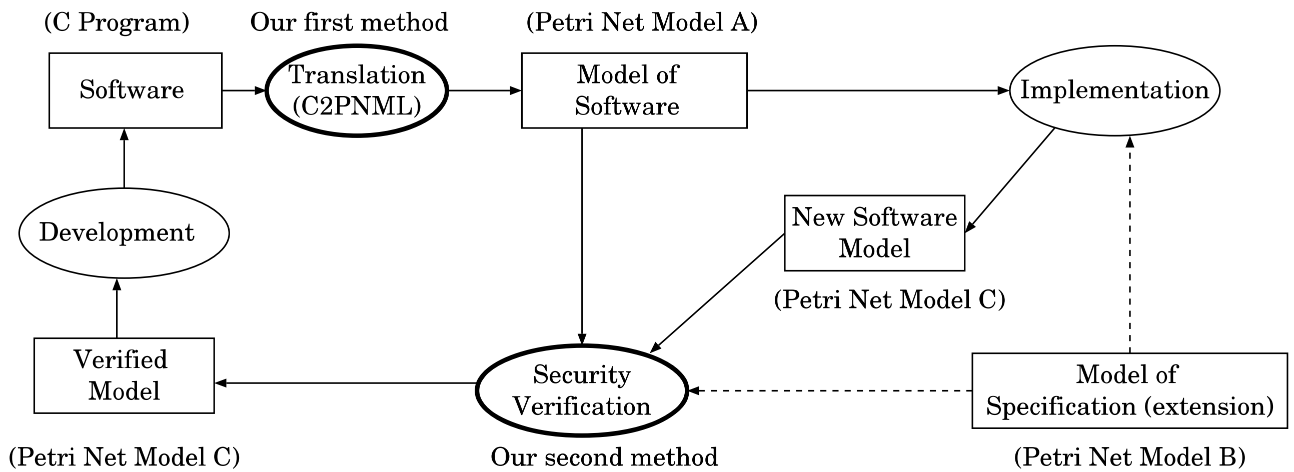


Figure 3.17: Our proposed model-driven approach

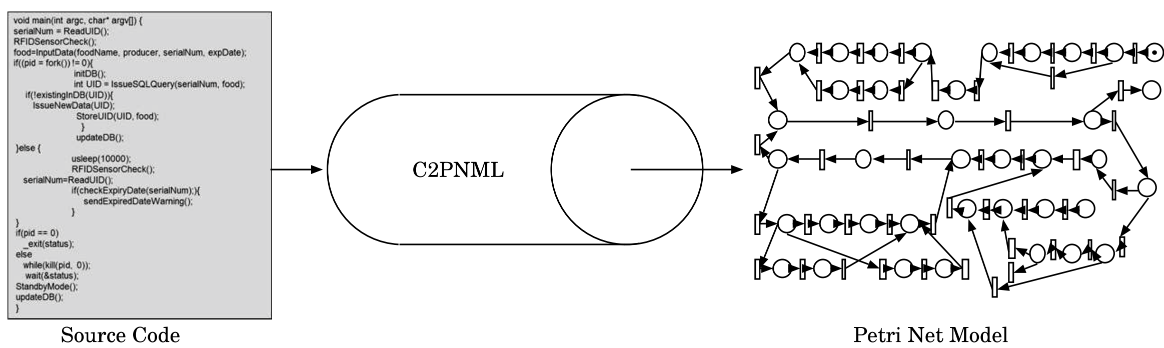


Figure 3.18: Our proposed translation approach

Petri net for further analysis. Generally, Petri net can be described with Petri Net Markup Language (PNML for short). C2PNML will analyze the syntax of the program and convert the program into Petri net by constructing place and transitions (see Fig. 3.19). Finally, C2PNML will output the PNML file. In Fig. 3.19, there are two translation procedures. First, the user procedure is only to use C2PNML as a tool and produce the required PNML file from C program. Second, in the developer procedure, when additional syntax or statements are required, we can extend C2PNML by adding new expression statement and parsing table. C2PNML can convert statements in the the whole program not including functions. We can translate a function inner source code into Petri net and refine the Petri net of the function into the Petri net of the main function in the program. We propose the following translation procedure:

≪Program to Petri Net's Translation≫

Input: C Program π

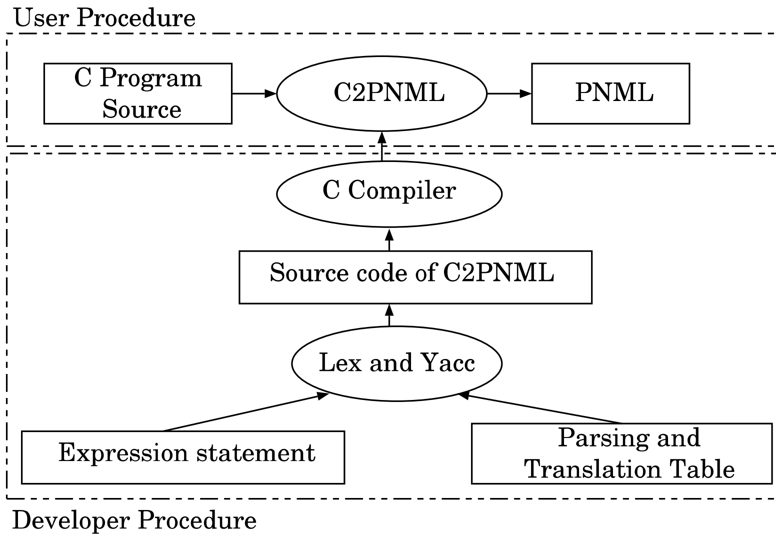


Figure 3.19: Structure of C2PNML.

Output: Incidence Matrix $\mathbf{A}$ of π's control flow and initial marking vector M_0

1° Generate matrix $\mathbf{N}\leftarrow\mathbf{O}_{PMAX\times TMAX}$ ($PMAX$ is the maximum number of places and $TMAX$ is the maximum number of transitions), vector $\mathbf{L}\leftarrow(\overbrace{1,0,0,\cdots,0}^{PMAX})$, $p\leftarrow1$ and $t\leftarrow1$, variable $x\leftarrow0$, empty stack S.

2° Analyse the syntax of π. Based on the conversion table in Table 3.1 set the value of $N(p,t)$.

3° Output $\mathbf{N}_{p\times t}$ as $\mathbf{A}$ and $\mathbf{L}_{p\times1}$ as M_0, and stop.

We implement C2PNML by combining the use of lexical analyser Lex and a parser generator Yacc [47] (see Fig. 3.19). We applied ANSI C rule to the lexical and syntax rules [48]. Then, we apply the actions in Table 3.1. Concretely, directly after `expression_statement` we describe the action as $N(p,t)\leftarrow-1, N(p+1,t)\leftarrow1, p\leftarrow p+1, t\leftarrow t+1$. Currently, C2PNML can handles (i) `if`, (ii) `for`, (iii) `while`, (iv) `fork`, (v) semaphores P operation `sem_wait` and (vi) V operation `sem_post`. By enriching the conversion actions in Table 3.1, more syntax can be handled with C2PNML.

The rules in Table 3.1 can convert parallel structured programs. The parallel structured program includes sequence, parallel, choice and loop construct. The structure patterns is shown in Fig. 3.24.

We take an example as shown in Fig. 3.25. By using the ≪Program to Petri Net's Conversion Procedure≫ we convert the program π_2 in Fig. 3.25 into Petri net model. The example of

Table 3.1: Conversion from C program to Petri Net

Syntax	Petri Net	
`expression_statement{a}`	a	$N(p,t) \leftarrow -1$ $N(p+1,t) \leftarrow 1$ $p \leftarrow p+1$ $t \leftarrow t+1$

`IF{a1}'('expression')'`
`statement{a2}`
`ELSE{a3}statement{a4}`
	a1	$N(p,t) \leftarrow -1$
		$N(p+1,t) \leftarrow 1$
		$push(p)$
		$p \leftarrow p+1$
		$t \leftarrow t+1$

	a2	$N(p,t) \leftarrow -1$ $N(p+1,t) \leftarrow 1$ $x \leftarrow pop()$ $p \leftarrow p+1$ $t \leftarrow t+1$ $push(p)$
	a3	$N(x,t) \leftarrow -1$ $N(p+1,t) \leftarrow 1$ $p \leftarrow p+1$ $t \leftarrow t+1$
	a3	$x \leftarrow pop()$ $N(p,t) \leftarrow -1$ $N(p+1,t) \leftarrow 1$ $N(x,t) \leftarrow 1$ $N(x,t+1) \leftarrow -1$ $N(p+1,t+1) \leftarrow 1$ $p \leftarrow p+1$ $t \leftarrow t+1$

conversion for program π_2 is illustrated in Fig. 3.26. We initialize `PMAX=TMAX=2000`. In step 1°, we generate the variables matrix $\mathbf{N} \leftarrow \mathbf{0}_{2000\times 2000}$ vector $\mathbf{L} \leftarrow (\overbrace{1,0,0,\cdots,0}^{2000})$ variable $p \leftarrow 1$ and $t \leftarrow 1$ $x \leftarrow 0$ we generate the empty stack S. In step 2°, we analyse the syntax of π_2 based on the conversion table in Table 3.1 set the value of $N(p,t)$. For example, in the program π_2 in Line 4 `scanf("%lf", &x);` is an expression statement we carry out $N(p,t) = N(1,1) \leftarrow -1$,$N(p+1,t) = N(2,1) \leftarrow 1$, $p \leftarrow p+1 = 2$, $t \leftarrow t+1 = 2$. $N(1,1) \leftarrow -1$ denotes connect the arc from

```
#include <stdio.h>
int main()
{
  Statement A
  Statement B
}
```

Figure 3.20: An example of a sequence construct.

```
#include <stdio.h>
int main()
{
  if processID == (fork () <0){
        Statement A
  }else Statement B
}
```

Figure 3.21: An example of a parallel construct.

```
#include <stdio.h>
int main()
if ( Condition )
          Statement B
  else Statement C
}
```

Figure 3.22: An example of a choice construct.

place p_1 to transition t_1, while $N(p+1,t) = N(2,1) \leftarrow 1$ denotes connects the arc from transition t_1 to place p_2. $p \leftarrow p+1 = 2$ and $t \leftarrow t+1 = 2$ denotes that the next target place to be set is p_2 and t_2. Finally, step 3° outputs the incidence matrix $\mathbf{A}_1$ of N_{π_2} as follows:

```
#include <stdio.h>
int main()
{
  for/while ( Statement A ){
        Statement B
  }
}
```

Figure 3.23: An example of a loop construct.

(a) Sequence

(b) Parallel

(c) Choice

(d) Loop

Figure 3.24: The patterns for parallel structured programs.

$$
\mathbf{A_1} = \begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \\ p_9 \end{array}
\begin{array}{c} \begin{array}{ccccccccc} t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \end{array} \\
\begin{pmatrix}
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix} \end{array}
$$

```
1 main() {
2   double a;
3   int b;
4   scanf("%lf", &a);
5   if((int)a > a) {
6     b = (int)(a) - 1;
7   }
8   else {
9     b = (int)(a);
10  }
11  printf("%d\n", b);
12 }
```

Figure 3.25: A simple example of a C program π_2.

Initial marking $\mathbf{M}_1$ of incidence matrix $\mathbf{A}_1$ is as follows:

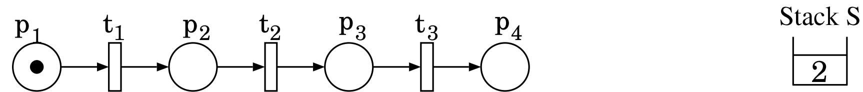
$$\mathbf{M_1} = \begin{pmatrix} \overset{p_1}{1} & \overset{p_2}{0} & \overset{p_3}{0} & \overset{p_4}{0} & \overset{p_5}{0} & \overset{p_6}{0} & \overset{p_7}{0} & \overset{p_8}{0} & \overset{p_9}{0} \end{pmatrix}$$

The proposed ≪Program to Petri Net's Translation≫ procedure enables us to reverse engineer a program into Petri net model so we can grasp the structure of the program. In general, translating the design model to source code is easy but to reverse the translation process is difficult. We reduce the effort to identify syntax in large program by proposing our tool C2PNML. C2PNML analyzes the syntax and statements then outputs the incidence matrix that represents the Petri net model. This significantly reduces the procedure taken to identify complex structures. The procedure can also runs in polynomial time which only takes $O(|P||T|)$ for the computation time. The syntax rule shown in Table 1 is highly customizable in order to adapt to the rule of the program. We only need to enrich the syntax rule in Table 1 in order to analyze various programs. Thus our tool is also available for real world applications.

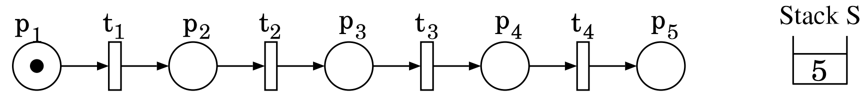
(a) Initial state.

(b) State after conversion at line 4 '`scanf("%lf", &a);`'

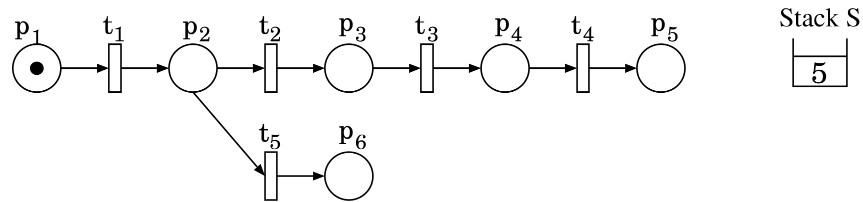
(c) State after conversion at line 5 '`if((int)x > x) {`'



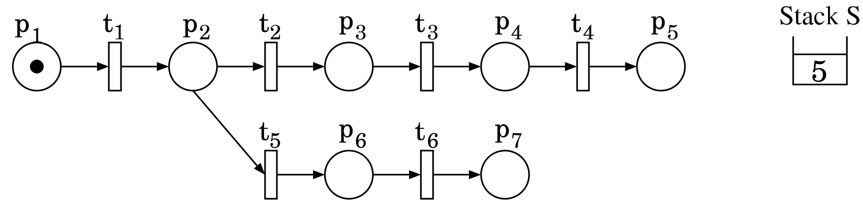
(d) State after conversion at line 6 '`y = (int)(x) - 1;`'



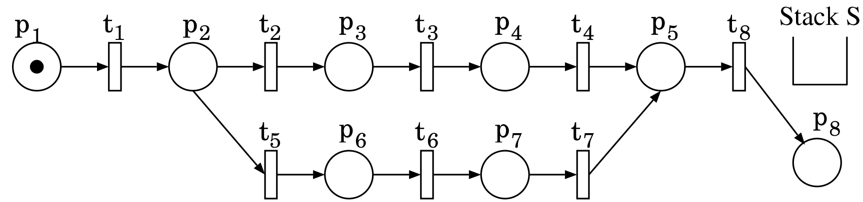
(e) State after conversion at line 7 (`if` statement '`}`'



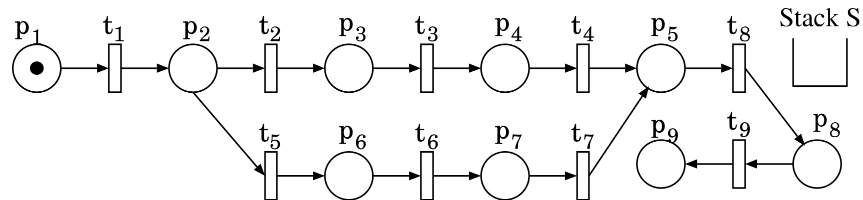
(f) State after conversion at line 8 '`else {`'



(g) State after conversion at line 9 '`b = (int)(a);`'



(h) State after conversion at line 10 (`else` statement '`}`'



(d) State after conversion at line 11 '`printf("%d\n", b);`'(the final state N_{π_2}).

Figure 3.26: Procedure execution for program π_2.

3.5 Representation of Program Structure with WF-net

Once we converted a program into WF-net. We can analyze its structure. First, we show the class of WF-nets that can represent program structure. We found that program structure can be represented with bridge-less WF-net. Therefore we give the property of bridge-less WF-net.

A Petri net can be divided into structural objects called cycles, handles, and bridges. These objects enable us to develop methods to structurally analyze the behavior of Petri nets.

Many actual computer programs can be modeled as a subclass of Petri nets without bridges, named bridge-less WF-nets. In 2013, Taniguchi et al. [49] have proposed a method to convert a C program to a bridge-less WF-net. Bin Ahmadon et al. [24] also have proposed a new method to analyze bridge-less WF-nets. This method converts a given bridge-less WF-net into a tree representation called process tree [?], and enables us to analyze the behavior of the net through the process tree. We, however, know little about properties on bridge-less WF-nets.

In this section, we reveal properties of bridge-less WF-nets, and give an application of bridge-less WF-nets to software design [50].

A WF-net N is said to be bridge-less if $\overline{N}$ has no bridge. We find out properties of bridge-less WF-nets.

Let us first focus on acyclic WF-nets.

Property 3.1 *Any acyclic bridge-less WF-net N is FC.* □

Proof: We prove the contraposition. We assume that N is not an FC WF-net. N has a structure shown in the dotted box in Fig. 3.27. For two nodes v and u in N, $v \overset{N}{\to} u$ denotes that u is reachable from v in N. There exist a node x such that $p_I \overset{N}{\to} x \overset{N}{\to} p_1$ and $p_I \overset{N}{\to} x \overset{N}{\to} p_2$; and a node y such that $t_1 \overset{N}{\to} y \overset{N}{\to} p_O$ and $t_2 \overset{N}{\to} y \overset{N}{\to} p_O$. This implies $x \overset{\overline{N}}{\to} p_2 \overset{\overline{N}}{\to} t_2 \overset{\overline{N}}{\to} y \overset{\overline{N}}{\to} p_O \overset{\overline{N}}{\to} t^* \overset{\overline{N}}{\to} p_I \overset{\overline{N}}{\to} x$. So $\overline{N}$ has a cycle $x \cdots p_2 t_2 \cdots y \cdots p_O t^* p_I \cdots x$ and a handle $x \cdots p_1 t_1 \cdots y$. And $p_1 t_2$ is a bridge between them. Thus N is not bridge-less.

Q.E.D.

Let us consider an acyclic WF-net N_1 shown in Fig. 3.28. If we look at p_4 and p_5, $p_4 \overset{N_1}{\bullet} \cap p_5 \overset{N_1}{\bullet} \neq \emptyset$. However, $|p_4 \overset{N_1}{\bullet}|=1$ and $|p_5 \overset{N_1}{\bullet}| \neq 1$, so N_1 is not FC. $\overline{N_1}$ has a bridge $t_2 p_4 t_4$. Thus N_1 is not bridge-less.

Let us consider the relation between acyclic bridge-less WF-nets and soundness.

Property 3.2 *An acyclic bridge-less WF-net N is sound iff N is WS.* □

Proof: The "If" part: From Property 5 in Ref. [51], any acyclic WS WF-net is sound, so N is sound.

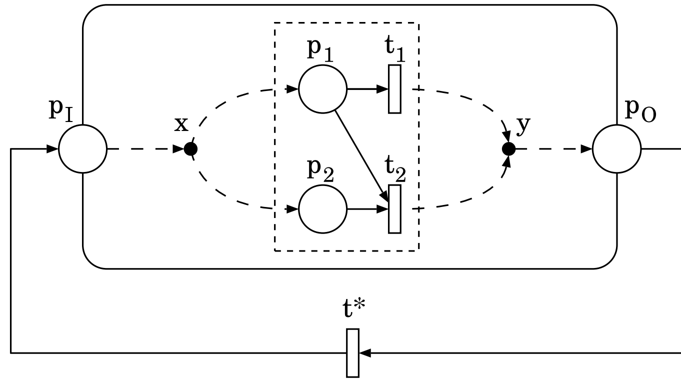


Figure 3.27: Illustration of the proof of Property 3.1. If an acyclic WF-net is in a class bigger than FC, its short-circuited net has a bridge.

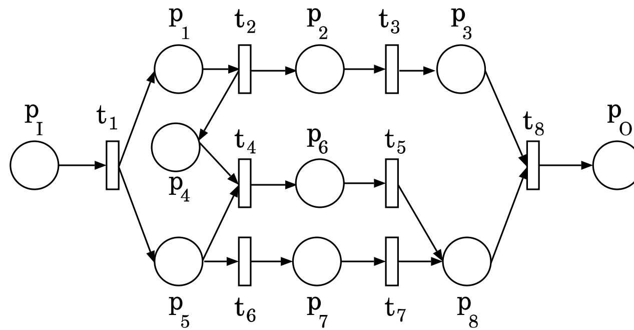


Figure 3.28: An acyclic WF-net N_1. N_1 is not bridge-less because $\overline{N_1}$ has a bridge $t_2 p_4 t_4$.

The "Only-if" part: Since N is bridge-less and non-WS, $\overline{N}$ has a TP-handle or a PT-handle which has no TP-bridge. From Property 3.1, $\overline{N}$ is a strongly-connected FC Petri net. From Theorem 4.2 in Ref. [52], if the strongly-connected FC Petri net has a TP-handle or a PT-handle which has no TP-bridge. the net has no live and bounded marking. This means that $\overline{N}$ has no live and bounded marking. Thus, N is not sound. **Q.E.D.**

This property means that soundness of acyclic bridge-less WF-nets can be decided by class. Figure 3.29 shows the illustration of Property 3.2.

Let us consider an acyclic WF-net N_2 shown in Fig. 3.30. N_2 is bridge-less because $\overline{N_2}$ has no bridge. $\overline{N_2}$ also has no TP-handle or PT-handle, so N_2 is WS. Thus N_2 is sound. Then let us consider an acyclic WF-net N_3 shown in Fig. 3.31. N_3 is bridge-less because $\overline{N_3}$ has no bridge. $\overline{N_3}$ has a TP-handle $t_1 p_4 t_4 p_2$, so N_3 is not WS but FC. Thus N_3 is not sound.

Next let us consider the whole of WF-nets including ones having cycles.

Property 3.3 *Any sound bridge-less WF-net is WS.* □

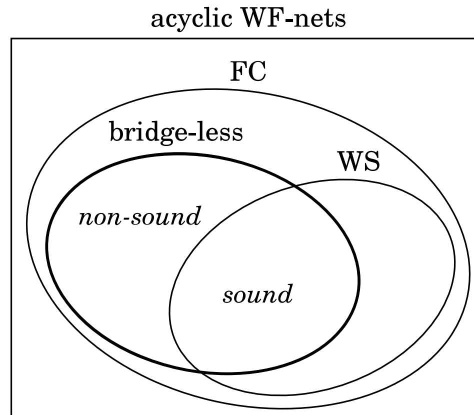


Figure 3.29: Acyclic bridge-less WF-nets and soundness.

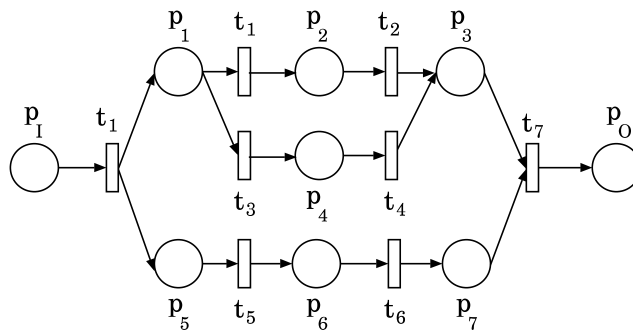


Figure 3.30: An acyclic WF-net N_2. N_2 is bridge-less WS. Thus N_2 is sound.

Proof: We prove that a sound WF-net N is not bridge-less if N is not WS. We assume that N is a sound non-WS WF-net. This implies that N is not a van Hee's ST-net [38]. Intuitively, ST-nets are constructed from SM WF-nets and acyclic MG tWF-nets [1] by means of refinement. Since N is not a ST-net, there exists a path ρ between a SM WF-net and an acyclic MG tWF-net in $\overline{N}$. The SM WF-net and the acyclic MG tWF-net are respectively denoted by N_{SM} and N_{MG}. From the construction of $\overline{N}$, $\overline{N}$ has a cycle through N_{SM} and a handle through N_{MG}. This implies that ρ becomes a bridge between the cycle and the handle. Thus N is not bridge-less. **Q.E.D.**

This property means a necessary condition for bridge-less of sound WF-nets, i.e. A sound WF-net N is not bridge-less if N is not WS.

Let us consider a cyclic WF-net N_4 shown in Fig. 3.32. N_4 is bridge-less and sound. Thus N_4 is WS.

[1]The dual nets [53] of WF-nets are called tWF-nets.

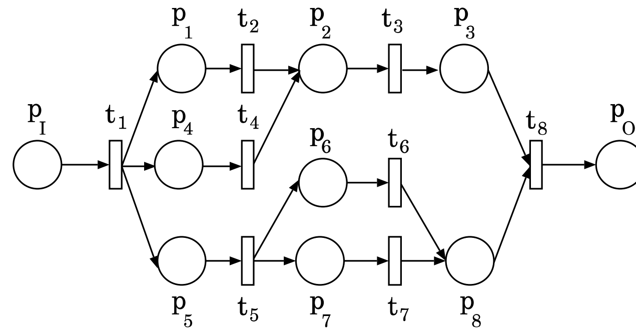


Figure 3.31: An acyclic WF-net N_3. N_3 is bridge-less FC. Thus N_3 is not sound.

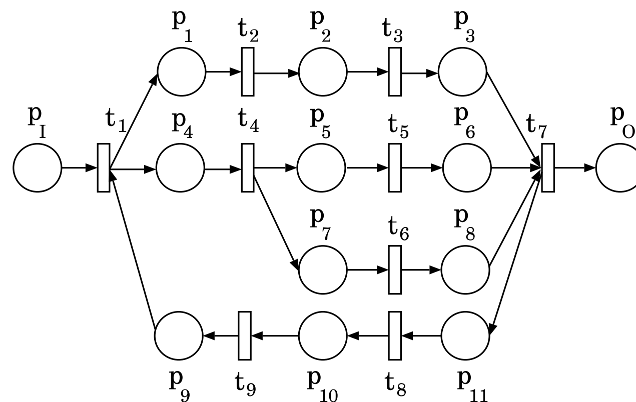


Figure 3.32: A (cyclic) WF-net N_4. N_4 is bridge-less and sound. Thus N_4 is WS.

3.5.1 Decision Method for Bridge-less of WF-nets

In this section, we tackle a problem of deciding whether a given WF-net is bridge-less. We first give a necessary and sufficient condition on the problem. Then we construct a polynomial-time procedure based on the condition.

3.5.2 Necessary and Sufficient Condition

We first restrict our analysis not to the whole of a WF-net but to a part between a node x and another node y of the net. Let us consider whether there is a handle from x to y which has a bridge b. If such a handle h exists, h and its cycle c are connected through b, i.e. the two disjoint paths corresponding to h and c form a single connected component. This enables us to give the following property.

Lemma 3.1 *Let N be a WF-net, and (x, y) a node pair in N. Let $\overline{N}'$ denote the Petri net obtained*

by removing the output arcs of x and the input arcs of y from $\overline{N}$ [2]*. There is a handle from x to y which has a bridge in $\overline{N}$ iff the following equation holds:*

$$\begin{aligned}&\textit{(the number of connected components in } \overline{N}'\textit{)}\\ &\neq \textit{(the number of disjoint paths from } x \textit{ to } y \textit{ in } \overline{N}\textit{)} + 1.\end{aligned} \tag{3.1}$$

□

Proof : The "If" part: We prove the contraposition: If there is no handle from x to y which has a bridge in $\overline{N}$ then Eq. (3.1) does not hold. Without such a handle, each handle (disjoint path) from x to y corresponds one-to-one to a connected component in $\overline{N}'$. The remains of $\overline{N}'$ form a single connected component. So we have (the number of connected components in $\overline{N}'$) = (the number of disjoint paths from x to y)+1. Thus Eq. (3.1) does not hold.

The "Only-if" part: If there is a handle h from x to y which has a bridge b in $\overline{N}$, h and its cycle c are connected through b. This implies that the two disjoint paths corresponding to h and c form a single connected component. So we have (the number of connected components in $\overline{N}'$) < (the number of disjoint paths from x to y)+1. Thus Eq. (3.1) holds. **Q.E.D.**

Extending Lemma 3.1 to the whole of the given WF-net, we can obtain the following necessary and sufficient condition for deciding whether the net is bridge-less.

Theorem 3.2 *A WF-net N $(=(P,T,A,\ell))$ is bridge-less iff for every node pair $(x,y)\in(P\cup T)^2$, the following equation holds:*

$$\begin{aligned}&\textit{(the number of connected components in } \overline{N}'\textit{)}\\ &\neq \textit{(the number of disjoint paths from } x \textit{ to } y \textit{ in } \overline{N}\textit{)} + 1,\end{aligned}$$

where $\overline{N}'$ is the Petri net obtained by removing the output arcs of x and the input arcs of y from $\overline{N}$. □

3.5.3 Polynomial-time Procedure

Based on the proposed necessary and sufficient condition, we construct a polynomial-time procedure for deciding whether a given WF-net is bridge-less.

≪Decision of Bridge-less≫

Input: WF-net N $(=(P,T,A,\ell))$

Output: Is N bridge-less?

[2] For a WF-net N $(=(P,T,A,\ell))$, $\overline{N}'=(P, T\cup\{t^*\}, A\cup\{(p_O,t^*),(t^*,p_I)\}\backslash\{x\}\times x\overset{N}{\bullet}\backslash\overset{N}{\bullet}y\times\{y\}, \ell\cup\{(t^*,\tau)\})$

1° Construct a flow network $D_{\overline{N}}$ isomorphic to $\overline{N}$, where every edge of $D_{\overline{N}}$ has capacity 1.

2° For each node pair $(x, y) \in (P \cup T)^2$, do the following steps to check Eq. (3.1):

2-1° ▹ Calculate the number n of disjoint paths from x to y.
Calculate the maximum value of the flow from x to y in $D_{\overline{N}}$. The value is the same as n.

2-2° ▹ Calculate the number m of the connected components in the Petri net $\overline{N}'$ obtained by removing the output arcs of x and the input arcs of y from $\overline{N}$.
Construct $\overline{N}'$, and obtain m by counting the connected components of $\overline{N}'$.

2-3° If $m \neq n + 1$ then output no and stop.

3° Output yes, and stop.

Property 3.4 *The following problem can be solved in polynomial time: Given a WF-net N, to decide whether N is bridge-less.* □

Proof : Algorithm ≪Decision of Bridge-less≫ can run in polynomial time. Step 2-1° takes $O((|P|+|T|)|A|^2)$ with Edmonds-Karp algorithm [54]. Step 2-2° takes $O(|P|+|T|+|A|)$ with DFS. Thus the whole of the algorithm takes $O((|P|+|T|)^3|A|^2)$. **Q.E.D.**

3.5.4 Example

We illustrate ≪Decision of Bridge-less≫ with two examples.

The first example is for a WF-net N_5 shown in Fig. 3.33. In Step 1°, we construct the flow network $D_{\overline{N_5}}$ isomorphic to $\overline{N_5}$, where every edge of $D_{\overline{N_5}}$ has capacity 1. In Step 2°, we check Eq. (3.1) for each node pair of N_5. For (p_1, p_2), we first calculate the number n of disjoint paths from p_1 to p_2. We can obtain $n = 2$ by calculating the maximum value of the flow from p_1 to p_2 in $D_{\overline{N_5}}$. Next we calculate the number m of the connected components in the Petri net $\overline{N_5}'$ obtained by removing the output arcs of p_1 and the input arcs of p_2 from $\overline{N_5}$. $\overline{N_5}'$ is shown in Fig. 3.34. We obtain $m = 3$ by counting the connected components of $\overline{N_5}'$. Since $m = 3 = n+1$, so we can say that there is no handle from p_1 to p_2 which has a bridge in $\overline{N_5}$. In the same way, for every node pair, we can make sure that there is no handle which has a bridge in $\overline{N_5}$. Thus ≪Decision of Bridge-less≫ outputs yes. In fact, N_5 is bridge-less.

The second example is for a WF-net N_6 shown in Fig. 3.35. In Step 1°, we construct $D_{\overline{N_6}}$. In Step 2°, we check Eq. (3.1) for each node pair of N_6. For (t_2, t_1), we first obtain $n = 2$ by

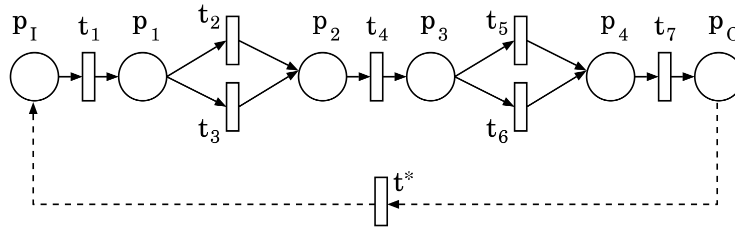


Figure 3.33: A WF-net N_5.

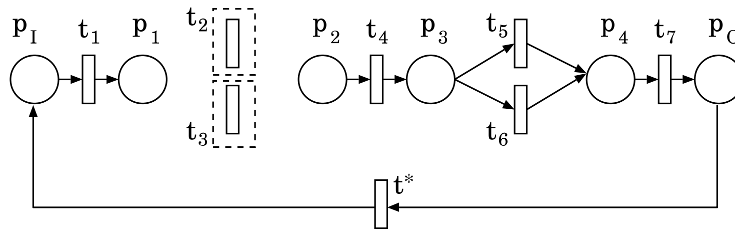


Figure 3.34: The Petri net $\overline{N_5}'$ obtained by removing the output arcs of place p_1 and the input arcs of place p_2 from $\overline{N_5}$. The number of connected components of $\overline{N_5}'$ is 3.

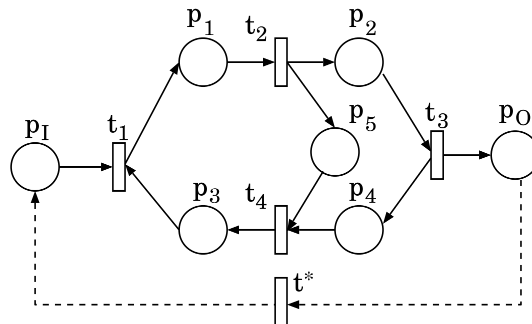


Figure 3.35: A WF-net N_6.

calculating the maximum value of the flow from t_2 to t_1 in $D_{\overline{N_6}}$. Next we obtain $m = 2$ by counting the connected components of $\overline{N_6}'$ shown in Fig. 3.36. Since $m = 2 \neq n+1 = 3$, so ≪Decision of Bridge-less≫ outputs no. In fact, there exists a bridge $t_3p_4t_4$ between a handle $t_2p_5t_4p_3t_1$ and its cycle $p_It_1p_1t_2p_2t_3p_Ot^*p_I$ in $\overline{N_6}$. Thus, N_6 is not bridge-less.

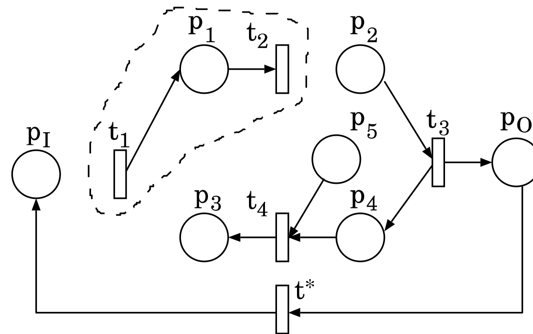


Figure 3.36: The Petri net $\overline{N_6}'$ obtained by removing the output arcs of place t_2 and the input arcs of place t_1 from $\overline{N_6}$. The number of connected components of $\overline{N_6}'$ is 2.

3.6 Remarks

In this chapter, we discussed the real world problem of software evolution. We know that during the evolution of software, it is important to manage and preserve the product line of the software. As software evolves to adapt to requirements of consumer over time, the newer version of the software tends to become more complex. We showed that we can analyzed software backward compatibility with Petri net. The, we proposed a reverse engineering method to translate program into Petri net model. We also showed that a class of Petri net can represent program structure called as bridge-less WF-net. We revealed the property for the Petri net class.

Chapter 4

Model-Driven Development in Software Evolution

In this chapter, after introducing our reverse engineering method in the previous chapter, we combine the method with our second major method. We proposed a model-driven verification method to solve conventional model checking problem in practical time. We tackle state number calculation in model checking, behavioral inheritance check and response property problem in a program. We utilized a representational bias called as process tree.

Figure 4.1 shows the overview of our model-driven development approach in detail. To verify whether a source code Y is backward compatible with its older version X, we first reverse engineer the code into Petri net model as shown in the previous chapter. In conventional analysis method, we use model checking approach by enumerating all states in the Petri net model. However, state space explosion can occur for analysis of large and complex program. Therefore, in our approach we first grasp the state number of the Petri net model. Then, if the state number is less than 1 million state conventional model checking approach is available. However, for state number more than 1 million states we propose our model-driven approach. Once, we grasp the state number we can proceed to the backward compatibility verification. For minor upgrade, we propose behavioral inheritance analysis. For major upgrade, we propose response property analysis. Then, our approach will output yes or no based on the verification result.

In order to be able to analyze the program in polynomial-time, we convert program into process tree. We proposed the convertibility and conversion algorithm for the conversion of Petri net to process tree [24]. We also show that the proposed algorithms and procedures are executable in polynomial time. Then we show an application example for each methods to illustrate the proposed approach.

In this chapter, we first convert a WF-net to process tree. We show how the state space can be

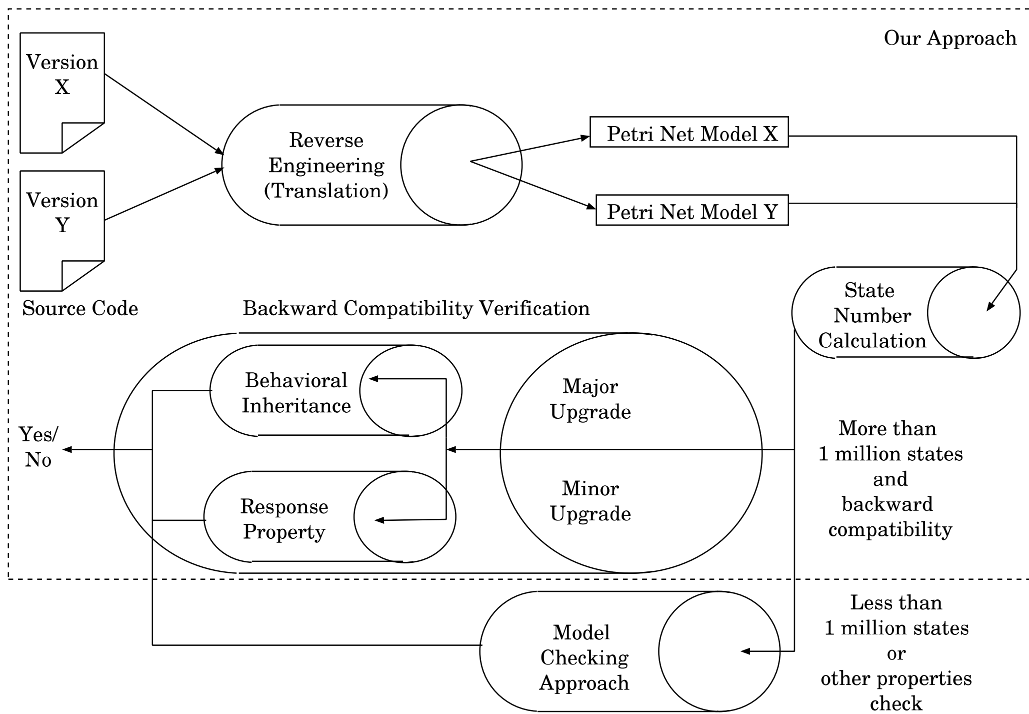


Figure 4.1: Overview in detail of our approach.

grasp with process tree such as state number calculation and then we show the details to verify backward compatibility without enumerating all states in the program. Concretely, we propose behavioral inheritance and response property analysis for minor and major upgrade. The verified model will be used to develop new version. In the new version, the backward compatibility is preserved by assuming the model is implemented properly into the new program.

4.1 Convertibility of Workflow Net to Process Tree

We can grasp the state space of a software but it is intractable but we cannot give up solving the problem because the problem is important for analysing program behavior. We try to utilize process trees to solve the state number calculation problem. The structure of process tree allows us to calculate state number without enumerating all states. Unfortunately, not all WF-nets are always convertible to process trees. For example, non-sound WF-nets are not convertible because the process tree itself is the representational bias of sound WF-net. Soundness is a necessary condition but is not sufficient. It is necessary to decide whether a given WF-net is convertible to a process tree or not. We call this problem as convertibility problem. In this section, we first give a formal definition of convertibility problem. Then we give a necessary and sufficient condition

on the problem.

4.1.1 Convertibility problem

We formalize convertibility problem as follows:

Definition 4.1 (Convertibility problem)
Instance : WF-net N
Question : Is N convertible to a process tree? □

Let us consider five instances of convertibility problem for example. The first instance is a WF-net N_1 shown in Fig. 4.2 (a). This WF-net can be represented as a process tree as shown in Fig. 4.3. By looking at Fig. 4.2 (a) we found that N_1 is an acyclic WS WF-net and has no bridge. The second instance is a WF-net N_2 shown in Fig. 4.2 (b). N_2 has a circuit $p_2t_2p_3t_3p_2$. In this paper, we use no operator representing circuits. Therefore, we assume that N_2 is not convertible to a process tree. The third instance is a WF-net N_3 shown in Fig. 4.2 (c). N_3 has a bridge $p_3t_7p_4$. Originally without the bridge (path $p_3t_7p_4$), paths $p_2t_2p_3t_3p_5$ and $p_2t_4p_4t_5p_5$ construct an exclusive choice but since bridge $p_3t_7p_4$ exists, $t_2p_3t_7p_4t_5$ forms a new sequence relation connecting the path. So actions t_2 and t_5 have two relations, an exclusive choice and a sequence. It is not convertible because one process tree operator can only represent one routing relation between actions. The forth instance is a WF-net N_4 shown in Fig. 4.2 (d). N_4 has a path $t_1p_2t_2p_6t_3p_8t_5$ with a handle $t_1p_3t_4p_7t_5$. There exists a path $t_1p_4t_8p_5t_3$ between the path and its handle. Path $t_1p_4t_8p_5t_3$ is similar to a bridge but is not exactly a bridge. We call it "pseudo-bridge". It is not convertible because without the pseudo-bridge, t_1 has a parallel relation with t_3, but since the pseudo-bridge $t_1p_4t_8p_5t_3$ exists, a new relation exists between t_1 and t_5. Since the action t_1 has more than one relation it cannot be represented with process tree operator. In this paper, we call a WF-net N as "bridge-less" if the short-circuited net of N includes neither bridges nor pseudo-bridges. The fifth instance is a WF-net N_5 shown in Fig. 4.2 (e). N_5 has a TP-handle $t_1p_2t_2p_4$ and a PT-handle $p_5t_5p_6t_7$. Since N_5 is not WS and there are no operator to represent TP-handle and PT-handle, it is not convertible. By generalizing the analysis result, we deduced that acyclic, bridge-less and WS structure plays a core role in the convertibility problem.

4.1.2 Necessary and Sufficient Condition

We propose a necessary and sufficient condition on the convertibility problem. For this we (i) define a subclass of WF-nets called as Process Tree Based (PTB for short) WF-net which can be

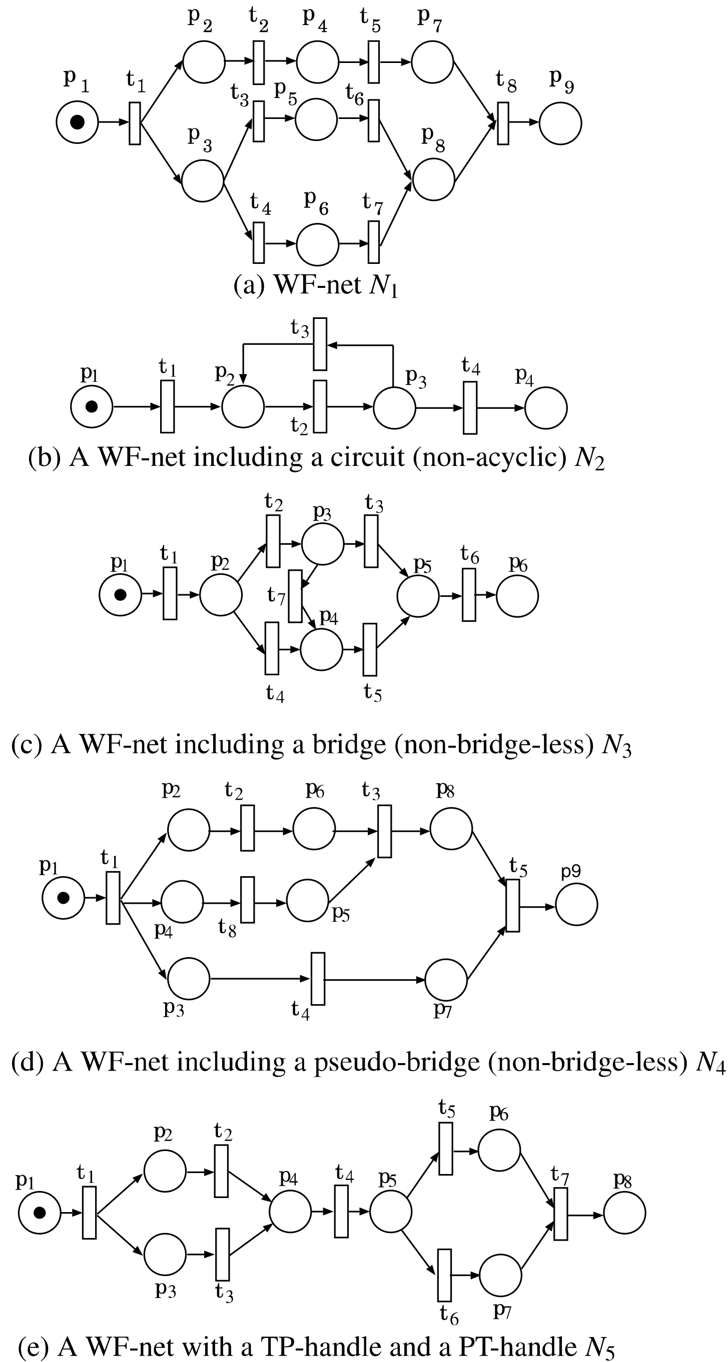


Figure 4.2: Example of WF-net instances.

represented as a process tree and (ii) show the PTB WF-net is acyclic, bridge-less and WS and (iii) show that a WF-net is PTB, i.e. convertible to a process tree iff it is acyclic, bridge-less and WS.

Definition 4.2 (PTB WF-net) *For any process tree π, let N be the WF-net itself and N_i (i=$1, 2, \cdots, n$)*

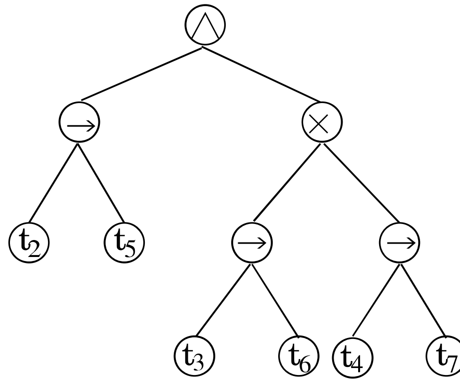


Figure 4.3: Process tree of N_1

be the subnet in N. Each p_I and p_O is the source place and the sink place of N, while each ${p_I}^{(n)}$ and ${p_O}^{(n)}$ is the source place and sink place of N_n. See Fig. 4.18 (the broken lines illustrate the boundaries of internal structure of WF-net N).

(i) If π is an action label, a WF-net N which consists of a transition representing the action label and its input and output places is PTB (See Fig. 4.18 (a)).

(ii) If π is $\oplus(\iota_1, \iota_2, \cdots, \iota_n)$ then let $N_1, N_2, \cdots, N_n$ be respectively PTB WF-nets representing action labels $\iota_1, \iota_2, \cdots, \iota_n$.

2-1° If $\oplus$ is sequence ($\rightarrow$) then a WF-net constructed by concatenating $N_1, N_2, \cdots, N_n$ which link the sink place of N_i with the source place of $N_{i+1}(1 \leq i < n)$ is PTB (See Fig. 4.18 (b)).

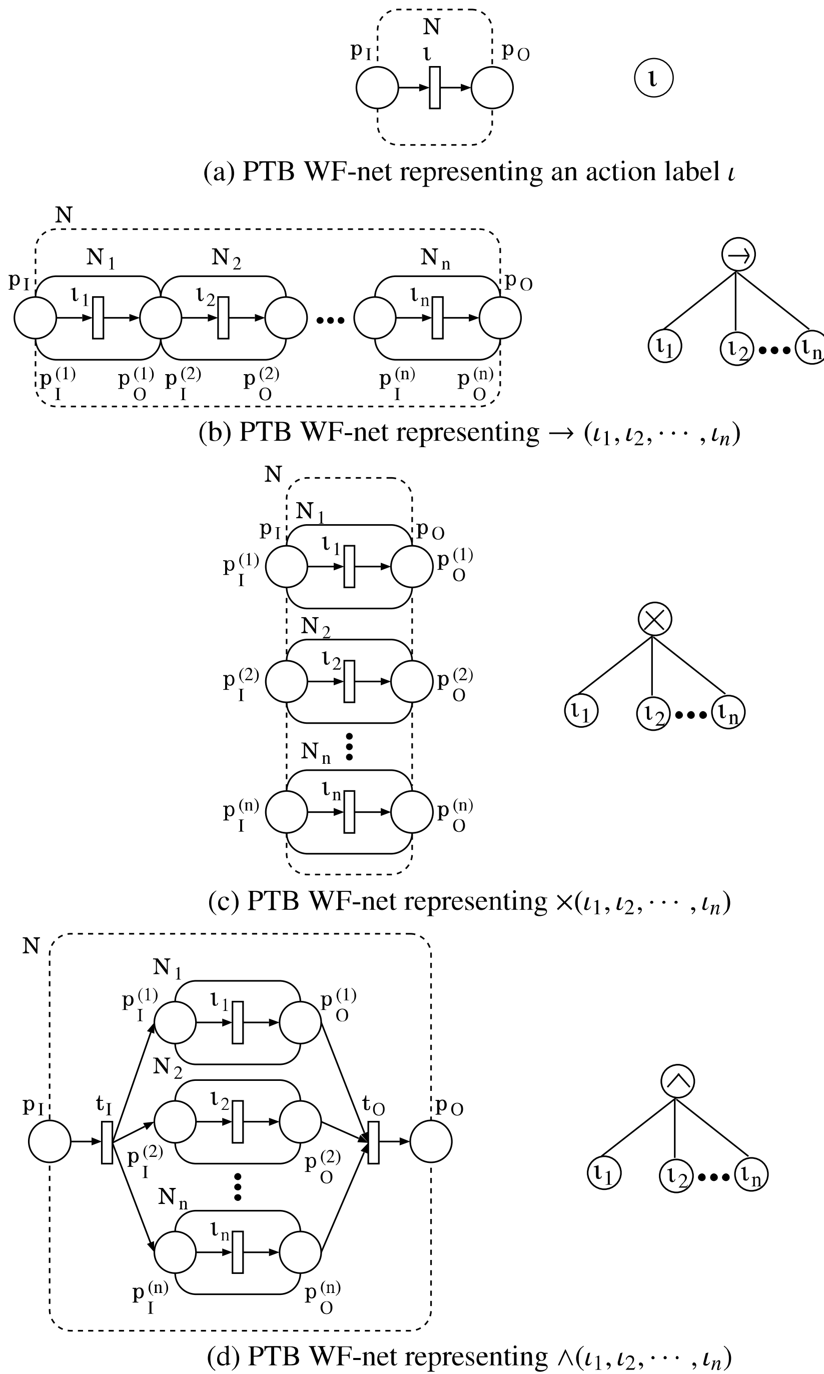
2-2° If $\oplus$ is exclusive choice ($\times$) then a WF-net constructed by bundling $N_1, N_2, \cdots, N_n$ which forms a selection of concurrent paths between their source places and sink places is PTB (See Fig. 4.18 (c)).

2-3° If $\oplus$ is parallel ($\wedge$) then a WF-net which is constructed by joining respectively all source places with a transition t_I, and sink places with a transition t_O of PTB WF-net $N_1, N_2, \cdots, N_n$ is PTB (See Fig. 4.18 (d)).

(iii) If π is $\oplus(\pi_1, \pi_2, \cdots, \pi_n)$ then let $N_1, N_2, \cdots, N_n$ be respectively PTB WF-nets representing sub-process trees $\pi_1, \pi_2, \cdots, \pi_n$.

3-1° If $\oplus$ is sequence then a WF-net constructed by concatenating $N_1, N_2, \cdots, N_n$ which link the sink place of N_i with the source place of $N_{i+1}(1 \leq i < n)$ is PTB.

3-2° If $\oplus$ is exclusive choice then a WF-net.



(a) PTB WF-net representing an action label ι

(b) PTB WF-net representing $\rightarrow(\iota_1, \iota_2, \cdots, \iota_n)$

(c) PTB WF-net representing $\times(\iota_1, \iota_2, \cdots, \iota_n)$

(d) PTB WF-net representing $\wedge(\iota_1, \iota_2, \cdots, \iota_n)$

Figure 4.4: Illustration of PTB WF-net and its equivalent process tree.

3-3° *If $\oplus$ is parallel then a WF-net constructed by joining respectively all source places with a transition t_I, and sink places with a transition t_O of PTB WF-net $N_1, N_2, \cdots, N_n$ is PTB.* □

N_1 shown in Fig. 4.2 (a) is PTB. Let us construct N_1 from the process tree shown in Fig. 4.3. For $\rightarrow(t_2, t_5)$ we construct a WF-net composed of a path $p_2t_2p_4t_5p_7$ based on Item (ii)-a) of Def. 4.2. For $\times(\rightarrow(t_3, t_6), \rightarrow(t_4, t_7))$ we constructed a WF-net by bundling paths $p_3t_3p_5t_6p_8$ and $p_3t_4p_6t_7p_8$ based on Item (iii)-b). We can obtain N_1 by bundling those WF-nets.

Lemma 4.1 *A WF-net is PTB iff N is acyclic, bridge-less and WS.* □

Proof: The proof of "if" part: We make use of van Hee et al. [38]'s ST-net. The set S of ST-net is the smallest set of nets N defined as follows: (i) If N is a WF-net then $N \in S$; (ii) If N is an acyclic MG WF-net then $N \in S$; (iii) If $N \in S$, p is a place in N, and $M \in S$ is a tWF-net then $N \otimes_p M \in S$; (iv) If $N \in S$, t is a transition in N, and $M \in S$ is a tWF-net then $N \otimes_t M \in S$. We show the following: (i) An acyclic bridge-less WS WF-net N is an ST-net. (ii) An acyclic, bridge-less ST-net is PTB.

We first show that an acyclic bridge-less WS WF-net N is an ST-net. Intuitively, ST-nets are constructed from SMs and MGs by means of refinement. Let $\mathcal{N}$ be a WF-net. Refinement of a place p in $\mathcal{N}$ with a WF-net $\mathcal{M}$ yields a WF-net, denoted by $\mathcal{N} \otimes_p \mathcal{M}$, built as follows: p is replaced in $\mathcal{N}$ by $\mathcal{M}$; transitions in $\overset{N}{\bullet} p$ become input transitions of the source place of $\mathcal{M}$, and transitions in $p \overset{N}{\bullet}$ become output transitions of the sink place of $\mathcal{M}$. Refinement of a transition t in $\mathcal{N}$ with a tWF-net $\mathcal{M}$ yields a WF-net, denoted by $\mathcal{N} \otimes_t \mathcal{M}$, built as follows: t is replaced in $\mathcal{N}$ by $\mathcal{M}$; places in $\overset{N}{\bullet} t$ become input places of the source transition of $\mathcal{M}$, and places in $t \overset{N}{\bullet}$ become output places of the sink transition of $\mathcal{M}$. The dual nets [53] of WF-nets are called tWF-nets. From the definition of WS, there are neither TP-handles nor PT-handles of any circuit in $\overline{N}$. This implies that $\overline{N}$ consists of a circuit c, PP-handles of c, and TT-handles of c. Any PP-handle includes both terminal nodes of a TT-handle, or includes none. We can look for an SM WF-net $\mathcal{M}$ as a subnet of N, which consists of PP-handles not including terminal nodes of any TT-handle. This implies $N = \mathcal{N} \otimes_p \mathcal{M}$ for some place p of a WF-net $\mathcal{N}$. Similarly, any TT-handle includes both terminal nodes of a PP-handle, or includes none. We can look for an acyclic MG tWF-net $\mathcal{M}$ as a subnet of N, which consists of TT-handles not including terminal nodes of any PP-handle. This implies $N = \mathcal{N} \otimes_t \mathcal{M}$ for some transition t of a WF-net $\mathcal{N}$. Repeating these refinements, we can show that N is an ST-net.

Next we show that an acyclic bridge-less ST-net N is PTB. Any acyclic bridge-less SM or MG WF-net is obviously PTB. Let $\mathcal{N}$ be a PTB WF-net, t a transition in $\mathcal{N}$ and $\mathcal{M}$ a acyclic bridge-less MG tWF-net. Let $\mathcal{M}'$ be a WF-net obtained by extending a place to each source transition and sink transition in $\mathcal{M}$. Since $\mathcal{N}$ and $\mathcal{M}'$ are PTB they have process trees $\pi_{\mathcal{N}}$ and $\pi_{\mathcal{M}'}$. $\mathcal{N} \otimes_t \mathcal{M}$ has a process tree by replacing transition t in $\pi_{\mathcal{N}}$ with $\pi_{\mathcal{N}}$. Therefore $\mathcal{N} \otimes_t \mathcal{M}$ is PTB.

In the similar way, $\mathcal{N} \otimes_p \mathcal{M}$ is PTB.

The proof of "only if" part: If a WF-net is PTB, then it is acyclic bridge-less WS. From Item (ii)-a) of Def. 4.2 $\rightarrow(\iota_1, \iota_2, \cdots, \iota_n)$ constructs an WF-net which is a path. It is acyclic, bridge-less and WS. From Item (ii)-b) of Def. 4.2 $\times(\iota_1, \iota_2, \cdots, \iota_n)$ constructs an acyclic bridge-less SM WF-net. It is WS. From Item (ii)-c) of Def. 4.2 $\wedge(\iota_1, \iota_2, \cdots, \iota_n)$ constructs an acyclic bridge-less MG WF-net. It is WS. From Item (iii) of Def. 4.2, for each operator $\oplus$, $\oplus(\pi_1, \pi_2, \cdots, \pi_n)$ constructs a WF-net obtained by combining acyclic bridge-less WS WF-nets. Therefore the obtained WF-net is also acyclic, bridge-less and WS. **Q.E.D.**

Theorem 4.1 *A WF-net N is convertible to a process tree iff N is acyclic, bridge-less and WS.* □

This theorem means the necessary and sufficient condition on the convertibility problem. Any acyclic WS WF-net is sound [51], so all the acyclic bridge-less WS WF-nets are sound. This coincides with van der Aalst's necessary condition on convertibility. All acyclic WS WF-nets, however, cannot always be converted to process trees because some of them have bridges. This is the difference between van der Aalst's necessary condition and our necessary and sufficient condition.

By using the necessary and sufficient condition, let us decide whether N_1 shown in Fig. 4.2 (a) is PTB. N_1 is acyclic bridge-less WS. So N_1 is PTB i.e. convertible to a process tree.

Lemma 4.2 *The following problem can be solved in polynomial time: Given a WF-net N, to decide whether N is PTB.* □

Proof: We only have to show that each condition of Theorem 4.1 can be checked in polynomial time. Acyclicity is obviously decidable in polynomial time (See Ref. [55]). Bridge-less property can also be decided in polynomial time by searching for nodes connecting two parallel paths and handles that does not split and join at the same nodes (See Ref. [50]). We can also decide in polynomial time whether a given WF-net is WS by applying a modified version of the max-flow min-cut technique [34]. **Q.E.D.**

4.1.3 Conversion Algorithm of Workflow Net to Process Tree

We can implement the given procedure with the following algorithm. The difference of the modified DFS with regular DFS is that, if all input node $\bullet n$, is not finished, it will backtrack to the node which has unfinished node(s). In regular DFS, the algorithm will proceed to the next

unvisited node whether all its input node are visited or not. In the conversion algorithm, the input and output node is important to decide the position of node n in the routing.

≪Process Tree Conversion Algorithm≫

Input: PTB WF-net, $(N, [p_I])$
Output: Process tree formula, f_π

MAKEPROCESSTREE(N, π)
1 $f_\pi \leftarrow \varepsilon$
2 **for each** $n \in P \cup T$
3   color(n) ← *white*
4 VISITNODE(p_I)
5 Output f_π, and stop

VISITNODE(n)
1 **if** $n \in T$ **then**
2   **if** $|n\bullet| \geq 2$ **then**
4   **if** $|\bullet n|=1$ and input of n is p_I **then**
3       $f_\pi \leftarrow f_\pi$ + '→ (' + 'n' + '∧('
4   **else**
5       $f_\pi \leftarrow f_\pi$ + 'n' + '∧('
6   **if** $|\bullet n|=1$ and $|n\bullet|=1$ **then**
7   p ← the input node of n, q ← the output node of n
8   **if** $|p\bullet| \geq 2$ **then**
9       $f_\pi \leftarrow f_\pi$ + '→ (' + 'n'
10   **else if** $|\bullet q| \geq 2$ **then**
11       $f_\pi \leftarrow f_\pi$ + 'n' + ')'
12   **else if** $p=p_I$ and $q \neq p_O$ **then**
13       $f_\pi \leftarrow f_\pi$ + '→ (' + 'n'
14   **else**
15       $f_\pi \leftarrow f_\pi$ + 'n'
16   **if** $\forall u \in \bullet n$: color(u) is *black* **then**
17   color(n) ← *black*
18   **if** $|\bullet n| \geq 2$ and $|n\bullet| \geq 2$ **then**

19 $f_\pi \leftarrow f_\pi$ + ‘)’ + ‘n’ + ‘$\rightarrow$ ($\wedge$(’
20 **else if** $|\bullet n| \geq 2$ **then**
21 $f_\pi \leftarrow f_\pi$ + ‘)’ + ‘n’
22 **else**
23 **if** color(n) is not *gray*
24 color(n) $\leftarrow$ *gray*
25 **if** $n \in P$ **then**
26 **if** $|n\bullet| \geq 2$ **then**
27 $f_\pi \leftarrow f_\pi$ + ‘$\times$(’
28 **if** $|\bullet n|=1$ and $|n\bullet|=1$ **then**
29 $p \leftarrow$ the input node of n, $q \leftarrow$ the output node of n
30 **if** $|p\bullet| \geq 2$ and $|\bullet q|=1$ **then**
31 $f_\pi \leftarrow f_\pi$ + ‘$\rightarrow$ (’
32 **else if** $|\bullet q| \geq 2$ and $|p\bullet|=1$ **then**
33 $f_\pi \leftarrow f_\pi$ + ‘)’
34 **if** $\forall u \in \bullet n$: color(u) is *black* **then**
35 color(n) $\leftarrow$ *black*
36 **if** $|\bullet n| \geq 2$ and $|n\bullet| \geq 2$ **then**
37 $f_\pi \leftarrow f_\pi$ + ‘)’ + ‘n’ + ‘$\rightarrow$ ($\times$(’
38 **else if** $|\bullet n| \geq 2$ **then**
39 $f_\pi \leftarrow f_\pi$ + ‘)’
40 **if** $n=p_O$ **then**
41 $f_\pi \leftarrow f_\pi$ + ‘)’
42 **else**
43 **if** color(n) is not *gray*
44 color(n) $\leftarrow$ *gray*
45 **if** color(n) is *black* **then**
46 **for each** $t \in n\bullet$ **then**
47 VisitNode(t)

This procedure can run in polynomial time because it is based on DFS, as stated in the following property.

Theorem 4.2 *≪Conversion of WF-net model to process tree≫ runs in polynomial time.* □

Proof : *≪Process Tree Conversion Algorithm≫ runs in polynomial time because the DFS takes time* $O(|P|+|T|+|A|)$.

Next, an example of a conversion of PTB WF-net N to process tree f_π is shown in the next sub-section.

4.1.4 Example of Conversion

The given WF-net N_1 and N_9 is known to be a PTB WF-net. We take both WF-nets to be converted to process tree.

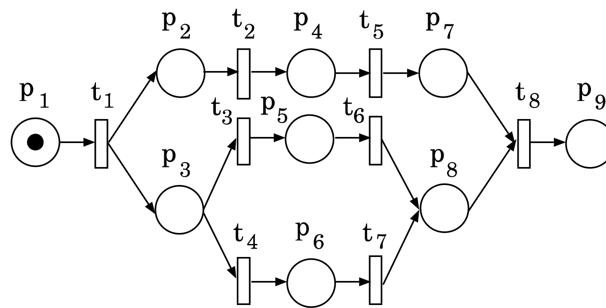


Figure 4.5: PTB WF-net N_1

The proposed algorithm with an application example to PTB WF-net N_1 is shown in Fig. 4.5. Table 4.2 shows the table for each node visited and the process tree formula f_π at certain node progress. All nodes are recursively searched and only transition nodes is stacked into f_π. The procedure is recursively repeated until $n\bullet = \emptyset$. The process tree for PTB WF-net in Fig. 4.5 can be represented as $\rightarrow (t_1 \wedge (\rightarrow(t_2, t_5), \times(\rightarrow(t_3, t_6), \rightarrow(t_4, t_7)))t_8)$. The process tree diagram is shown in Fig. 4.3.

Table 4.1: Algorithm execution at each nodes of N_1

P	T	$\lvert V_I \rvert$	$\lvert V_O \rvert$	Extend to f_π
p_1		0	1	
	t_1	1	2	$\rightarrow(t_1 \wedge($
p_2		1	1	$\rightarrow($
	t_2	1	1	t_2
p_4		1	1	
	t_5	1	1	t_5
p_7		1	1	$)$
	t_8	2	1	
p_3		1	2	$\times($
	t_3	1	1	$\rightarrow(t_3$
p_5		1	1	
	t_6	1	1	$t_6)$
p_8		2	1	
	t_4	1	1	$\rightarrow(t_4$
p_6		1	1	
	t_7	1	1	$t_7)$
p_8		2	1	$)$
	t_8	2	1	$)t_8$
p_9		1	0	$)$

Conversion execution of PTB WF-net N_9 is shown in Fig. 4.6 is shown in Table 4.2. The WF-net can be represented as $\rightarrow(t_1 \wedge (\rightarrow(t_2, t_3) \rightarrow(t_4, t_5, t_6)t_7)t_8 \times (t_9, t_{10})t_{11})$. The process tree diagram is shown in Fig. 4.3.

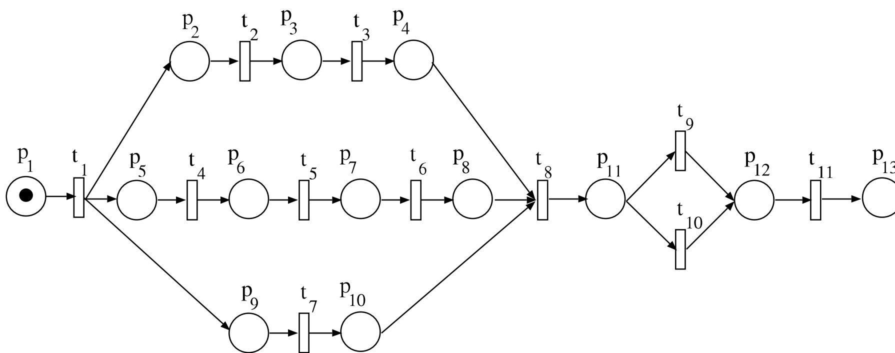


Figure 4.6: PTB WF-net N_9

Table 4.2: Algorithm execution at each nodes of N_9

P	T	$\lvert V_I \rvert$	$\lvert V_O \rvert$	Extend to f_π
p_1		0	1	
	t_1	1	3	$\to(t_1 \wedge ($
p_2		1	1	$\to($
	t_2	1	1	t_2
p_3		1	1	
	t_3	1	1	t_3
p_4		1	1	)
	t_8	3	1	
p_5		1	1	$\to($
	t_4	1	1	t_4
p_6		1	1	
	t_5	1	1	t_5
p_7		1	1	
	t_6	1	1	t_6
p_8		1	1	)
	t_8	3	1	
p_9		1	1	
	t_7	1	1	t_7
p_{10}		1	0	
	t_8	3	1	$)t_8$
p_{11}		1	1	$\times($
	t_9	1	1	t_9
p_{12}		1	2	
	t_{10}	1	1	t_{10}
p_{12}		1	2	)
	t_{11}	1	1	t_{11}
p_{13}		1	0	)

4.2 State Number Calculation

Petri net's state number is useful for analysis method that involves behavioural analysis such as in model checking approach. Model checking has been attracting attention as a promising approach to analysis of Petri nets. SPIN [20], a popular model checking tool, is available only to Petri nets with less than 1 million states, because SPIN basically enumerates all possible states in the Petri net. We need a polynomial time solution to accurately calculate the state number of the given Petri net before using SPIN.

In 2011, Chao et al. [56] proposed a method to calculate the number of all the possible

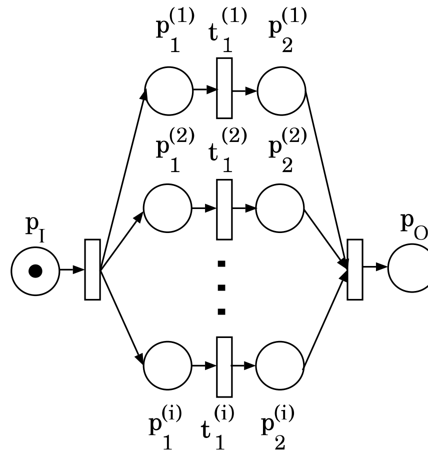


Figure 4.7: Illustration of a MG_i with (i+1) parallel paths. The state number is 2^i+2.

states. They first transformed a given Petri net to an algebraic expression, and then calculated the number of all the possible states by utilizing the algebraic expression. This method is, however, available only to a simple subclass i.e. MG and State Machine (SM for short). In addition, the computational complexity has not been discussed.

4.2.1 State Number Calculation Problem and Its Properties

In this section, we formalize a problem, named state number calculation problem [**?**], that calculates the number of all possible states in a given Petri net. Then we reveal the solvability and the computational complexity of the problem. The formal definition of the problem is given as follows:

Definition 4.3 (State number calculation problem)
Instance: Petri net (N, M_0)
Question: How many states are there in $R(N, M_0)$*?* □

As an example, in the case of WF-net of MG_i shown in Fig. 4.7, the problem is how many states there are in $R(MG_i, [p_I])$.

Let us consider the solvability of the state number calculation problem.

Property 4.1 *The state number calculation problem is solvable.* □

Proof: Let (N, M_0) be any Petri net. The state number calculation problem can be divided into two cases by the boundedness of (N, M_0). The boundedness problem is known to be decidable

[52]. If (N, M_0) is bounded, we have only to count the nodes in the reachability tree of (N, M_0). Otherwise, (if (N, M_0) is unbounded), we can regard $|R(N, M_0)|$ as ∞, where $\forall n \in \mathbb{N} : \infty > n$, $\infty \pm n = \infty$ and $\infty \geq \infty$. **Q.E.D.**

Let us consider the state number calculation problem of $(MG_i, [p_I])$ shown in Fig. 4.7. Since $(MG_i, [p_I])$ is bounded, we can solve the problem by using its reachability tree [32]. Unfortunately, $(MG_i, [p_I])$ has $2^i + 2$ markings. For example, to calculate the state number for the MG with i=20, we have to count 1,048,578 markings. In general, we cannot solve the problem by enumerating all the states in practical time.

Then, let us consider the computation complexity of the state number calculation problem. In this paper, we assume that P and NP are not equivalent, i.e $P \neq NP$. An NP-hard problem cannot be solved in polynomial time. We call the problem as intractable. We show that the state number calculation problem is intractable for FC WF-nets with initial marking $[p_I]$. To prove the intractability, we tackle the following decision version of the state number calculation problem: Given a Petri net (N, M_0), to decide whether $|R(N, M_0)| \geq \infty$. This decision problem is the boundedness problem. We have only to show that the boundedness problem is intractable for FC WF-nets with initial marking $[p_I]$.

To do so, we show that an NP-complete problem, called 3-conjunctive normal form boolean satisfiability problem [57] (3-CNF-SAT for short), can be transformed to the complement of the boundedness problem, i.e. the unboundedness problem of FC WF-nets with initial marking $[p_I]$.

Definition 4.4 (3-CNF-SAT [57])
Instance: Expression $\mathcal{E}$ of 3-conjunctive normal form that has n boolean variables and m clauses.
Question: Is there an assignment of variables satisfying $\mathcal{E}$=true? □

Lemma 4.3 The boundedness problem is co-NP-hard for FC WF-nets with initial marking $[p_I]$. □

Proof: We prove the co-NP-hardness by a reduction from 3-CNF-SAT in a way similar to Ref. [58]. Let $\mathcal{E}$ be an expression of 3-CNF-SAT which has n boolean variables $x_1, x_2, \cdots, x_n$ and m clauses $c_1, c_2, \cdots, c_m$. A literal ℓ_i is either a variable x_i or its negation $\overline{x_i}$. Without loss of generality, it can be assumed that $\mathcal{E}$ has all of x_i's and $\overline{x_i}$'s (i=$1, 2, \cdots, n$), and $m \geq 3$ [59]. We first construct the following Petri net $N_{\mathcal{E}}$=$(P_{\mathcal{E}}, T_{\mathcal{E}}, A_{\mathcal{E}})$.

$$P_{\mathcal{E}} = \{p_I, p_1, p_O\} \cup \bigcup_{i=1}^{n}\{q_i\} \cup \bigcup_{j=1}^{m}\{c_j\}$$
$$T_{\mathcal{E}} = \{t_1, t_2, t_3\} \cup \bigcup_{i=1}^{n}\{x_i, \overline{x_i}\}$$

$$\begin{aligned} A_{\mathcal{E}} = \ & \{(p_I, t_1), (t_2, p_1), (p_1, t_3), (t_3, p_1), (t_3, p_O)\} \\ & \cup \bigcup_{i=1}^{n} \{(t_1, q_i), (q_i, x_i), (q_i, \overline{x_i})\} \\ & \cup \bigcup_{k=1}^{3} \bigcup_{j=1}^{m} \{(\ell_k, c_j) | \ell_k \text{ is the } k\text{-th literal of clause } c_j\} \\ & \cup \bigcup_{j=1}^{m} \{(c_j, t_2)\} \end{aligned}$$

$N_{\mathcal{E}}$ is an FC WF-net because its short-circuited net $\overline{N_{\mathcal{E}}}$ is strongly connected; Places $c_1, c_2, \cdots, c_m$ share only one output transition t_2, and the other places share no output transition. $N_{\mathcal{E}}$ can be constructed in polynomial time, because it consists of (n+m+3) places, (2n+3) transitions, and (3n+4m+5) arcs.

Let us prove that $(N_{\mathcal{E}}, [p_I])$ is unbounded iff there is an assignment of variables satisfying $\mathcal{E}$=*true*.

The proof of "if" part: Let α denote an assignment of variables satisfying $\mathcal{E}$=*true*, and let $\ell_1, \ell_2, \cdots, \ell_n$ be the literals mapped to *true* by α. By the construction of $N_{\mathcal{E}}$, we have

$$\begin{aligned} [p_I] \quad & [N_{\mathcal{E}}, t_1\rangle \, [q_1, q_2, \cdots, q_n] \\ & [N_{\mathcal{E}}, \ell_1 \ell_2 \cdots \ell_n\rangle \, M \ (\geq [c_1, c_2, \cdots, c_m]). \end{aligned}$$

We are to show that $M \geq [c_1, c_2, \cdots, c_m]$. Since $N_{\mathcal{E}}$ is FC, we can freely choose, at every place q_i, between letting transition x_i or $\overline{x_i}$ fire. Since α satisfies $\mathcal{E}$, for each clause c_j ($1 \leq j \leq m$), there exists a literal ℓ_i ($1 \leq i \leq n$) in c_j. Therefore place c_j is marked by firing ℓ_i. As a result, we have

$$\begin{aligned} M \quad & [N_{\mathcal{E}}, t_2\rangle M' \ (= M \backslash [c_1, c_2, \cdots, c_m] \cup [p_1]) \\ & [N_{\mathcal{E}}, t_3\rangle M' \cup [p_O]. \end{aligned}$$

Since $M' \cup [p_O]$ covers M', $(N_{\mathcal{E}}, [p_I])$ is unbounded.

The proof of "only if" part: Let α denote any assignment of variables satisfying $\mathcal{E}$=*false*. Since α does not satisfy $\mathcal{E}$, there exists a clause c_j ($\in \{c_1, c_2, \cdots, c_m\}$) mapped to *false* by α. Let $\ell_1^j, \ell_2^j, \ell_3^j$ denote the literals in c_j. Since the corresponding transitions $\ell_1^j, \ell_2^j, \ell_3^j$ do not fire, their common output place, i.e. place c_j, is never marked. c_j is an input place of transition t_2, so t_2 is dead. This enables us to ignore the part following t_2 in $N_{\mathcal{E}}$. The remaining part is acyclic. Since any acyclic Petri net is bounded, $(N_{\mathcal{E}}, [p_I])$ is bounded.

Q.E.D.

For example, let us consider the following boolean expression:

$$\begin{aligned} \mathcal{E}_1 = \ & (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \\ & \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \\ & \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \end{aligned}$$

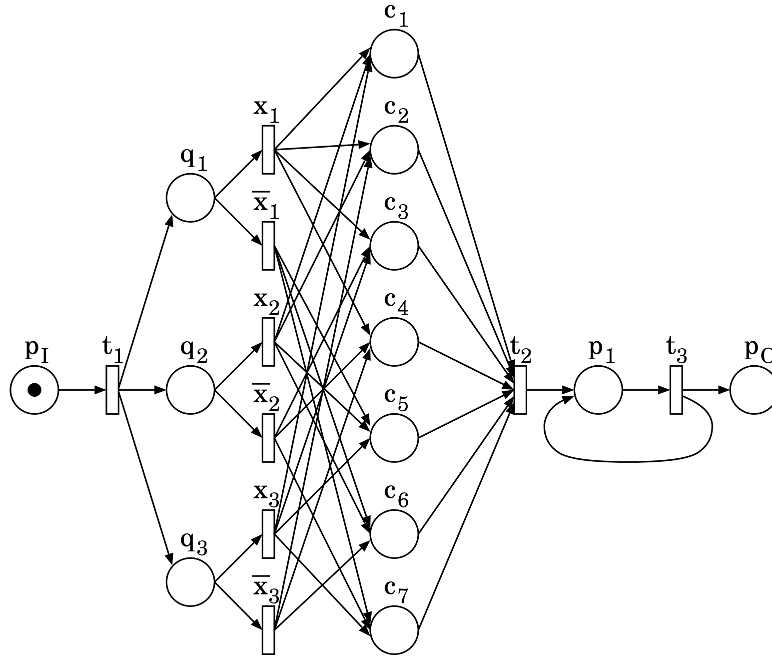


Figure 4.8: The FC WF-net $(N_{\mathcal{E}_1}, [p_I])$ corresponding to a 3-CNF-SAT expression $\mathcal{E}_1 = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$. $(N_{\mathcal{E}_1}, [p_I])$ is unbounded.

$\mathcal{E}_1$ is satisfiable by choosing x_1=*true*, x_2=*true*, x_3=*true*. Figure 4.17 shows the Petri net $N_{\mathcal{E}_1}$ constructed from $\mathcal{E}_1$. $(N_{\mathcal{E}_1}, [p_I])$ is unbounded, because

$$
\begin{aligned}
[p_I] \quad & [N_{\mathcal{E}_1}, t_1\rangle\, [q_1, q_2, q_3] \\
& [N_{\mathcal{E}_1}, x_1 x_2 x_3\rangle\, [{c_1}^3, {c_2}^2, {c_3}^2, c_4, {c_5}^2, c_6, c_7] \\
& [N_{\mathcal{E}_1}, t_2\rangle\, [{c_1}^2, c_2, c_3, c_5, p_1] \\
& [N_{\mathcal{E}_1}, t_3\rangle\, [{c_1}^2, c_2, c_3, c_5, p_1, p_O].
\end{aligned}
$$

From Lemma 4.3, we can obtain the following theorem.

Theorem 4.3 The state number calculation problem cannot be solved in polynomial time for FC WF-nets with initial marking $[p_I]$ if $P \neq NP$. □

Proof: The decision problem related to this problem, i.e. the boundedness problem, is co-NP-hard. This means that the original problem is intractable. **Q.E.D.**

4.2.2 Polynomial Time Procedure

In this section, we propose a polynomial time algorithm to calculate the state number by utilizing process tree.

Lemma 4.4 *Let N and π be respectively a PTB WF-net and its process tree. For each node v of π, $\pi(v)$ denotes the sub-tree of π whose root is v, $N(v)$ denotes the subnet of N represented as $\pi(v)$, and $s(v)$ denotes the number of possible states in $N(v)$.*

- *If v is a leaf node then*

$$s(v) = 2 \tag{4.1}$$

- *If v is an internal node then, let $v_1, v_2, \cdots, v_n$ be the children of v,*
  - *If v is sequence ($\rightarrow$) then*

$$s(v) = \sum_{i=1}^{n}(s(v_i) - 1) + 1 \tag{4.2}$$

  - *If v is exclusive choice ($\times$) then*

$$s(v) = \sum_{i=1}^{n}(s(v_i) - 2) + 2 \tag{4.3}$$

  - *If v is parallel ($\wedge$) then*

$$s(v) = \prod_{i=1}^{n} s(v_i) + 2 \tag{4.4}$$

□

Proof: If v is a leaf node, $N(v)$ is a PTB WF-net which consists of one transition and its input and output places. $N(v)$ is illustrated in Fig. 4.18(a). $(N(v), [p_I])$ has two states, $[p_I]$ and $[p_O]$, before and after the firing of the transition. Since $|R(N(v), [p_I])|$=2, we have $s(v) = 2$ =Eq. (4.1).

If v is sequence ($\rightarrow$), $N(v)$ is a PTB WF-net constructed by concatenating PTB WF-nets $N(v_1), N(v_2), \cdots$, and $N(v_n)$ so as to unite the sink place of $N(v_i)$ and the source place of $N(v_{i+1})$ ($1 \le i < n$). $N(v)$ is illustrated in Fig. 4.18(b). Let $p_I^{(i)}$ and $p_O^{(i)}$ denote respectively the source place and the sink place of $N(v_i)$. In $N(v)$, $[p_I]$ (=$[p_I^{(1)}]$) is reachable to $[p_O^{(1)}]$, $[p_I^{(2)}]$ (=$[p_O^{(1)}]$) is reachable to $[p_O^{(2)}]$, $\cdots$, $[p_I^{(n)}]$ (=$[p_O^{(n-1)}]$) is reachable to $[p_O^{(n)}]$ (=$[p_O]$), because $N(v_1)$, $N(v_2)$, $\cdots$, $N(v_n)$ is

sound. Since $N(v_i)$ and $N(v_{i+1})$ share only $p_O^{(i)}$ $(= p_I^{(i+1)})$, $(N(v_i), [p_I^{(i)}])$ and $(N(v_{i+1}), [p_I^{(i+1)}])$ have different states except $[p_O^{(i)}]$ $(= [p_I^{(i+1)}])$. Therefore we have

$$
\begin{aligned}
&R(N(v), [p_I]) \\
&\quad = (R(N(v_1), [p_I^{(1)}]) \backslash \{[p_O^{(1)}]\}) \cup \cdots \\
&\quad \cup (R(N(v_{n-1}), [p_I^{(n-1)}]) \backslash \{[p_O^{(n-1)}]\}) \cup R(N(v_n), [p_I^{(n)}]) \\
&|R(N(v), [p_I])| \\
&\quad = (|R(N(v_1), [p_I^{(1)}])| - 1) + \cdots \\
&\quad + (|R(N(v_{n-1}), [p_I^{(n-1)}])| - 1) + |R(N(v_n), [p_I^{(n)}])| \\
&s(v) = \sum_{i=1}^{n} (s(v_i) - 1) + 1 = \text{Eq. (4.2)}
\end{aligned}
$$

If v is exclusive choice $(\times)$, $N(v)$ is a PTB WF-net constructed by bundling PTB WF-nets $N(v_1), N(v_2), \cdots$, and $N(v_n)$ so as to unite respectively their source places and all their sink places. $N(v)$ is illustrated in Fig. 4.18(c). Note that $p_I = p_I^{(1)} = p_I^{(2)} = \cdots = p_I^{(n)}$ and $p_O = p_O^{(1)} = p_O^{(2)} = \cdots = p_O^{(n)}$. Since $N(v_1)$, $N(v_2)$, $\cdots$, and $N(v_n)$ share only the source places and the sink places, $(N(v_1), [p_I^{(1)}])$, $(N(v_2), [p_I^{(2)}])$, $\cdots$, and $(N(v_n), [p_I^{(n)}])$ have different states except $[p_I]$ $(= [p_I^{(1)}] = [p_I^{(2)}] = \cdots = [p_I^{(n)}])$ and $[p_O]$ $(= [p_O^{(1)}] = [p_O^{(2)}] = \cdots = [p_O^{(n)}])$. Therefore we have

$$
\begin{aligned}
&R(N(v), [p_I]) \\
&\quad = (R(N(v_1), [p_I^{(1)}]) \backslash \{[p_I^{(1)}], [p_O^{(1)}]\}) \cup \cdots \\
&\quad\quad \cup (R(N(v_n), [p_I^{(n)}]) \backslash \{[p_I^{(n)}], [p_O^{(n)}]\}) \cup \{[p_I], [p_O]\} \\
&|R(N(v), [p_I])| \\
&\quad = (|R(N(v_1), [p_I^{(1)}])| - 2) + \cdots \\
&\quad\quad + (|R(N(v_n), [p_I^{(n)}])| - 2) + 2 \\
&s(v) = \sum_{i=1}^{n} (s(v_i) - 2) + 2 = \text{Eq. (4.3)}
\end{aligned}
$$

If v is parallel $(\wedge)$, $N(v)$ is a PTB WF-net constructed by bundling PTB WF-nets $N(v_1), N(v_2)$, $\cdots$, and $N(v_n)$ so as to have another source place and another sink place. $N(v)$ is illustrated in Fig. 4.18(d). p_I is connected to $p_I^{(1)}$, $p_I^{(2)}$, $\cdots$, and $p_I^{(n)}$ via an additional transition t_I. This means that $[p_I][N(v), t_I\rangle[p_I^{(1)}, p_I^{(2)}, \cdots, p_I^{(n)}]$. Since $N(v_1)$, $N(v_2)$, $\cdots$, and $N(v_n)$ share no node, $(N(v_1), [p_I^{(1)}])$, $(N(v_2), [p_I^{(2)}])$, $\cdots$, and $(N(v_n), [p_I^{(n)}])$ have different states. Therefore $(N(v), [p_I^{(1)}, p_I^{(2)}, \cdots, p_I^{(n)}])$ has a combination of those states. Since $N(v_1)$, $N(v_2)$, $\cdots$, and $N(v_n)$ are sound, $[p_I^{(1)}, p_I^{(2)}, \cdots, p_I^{(n)}]$ is reachable to $[p_O^{(1)}, p_O^{(2)}, \cdots, p_O^{(n)}]$. $p_O^{(1)}$, $p_O^{(2)}$, $\cdots$, and $p_O^{(n)}$ are connected to p_O via another additional transition t_O. This means that $[p_O^{(1)}, p_O^{(2)}, \cdots, p_O^{(n)}][N(v), t_O\rangle[p_O]$. Therefore

we have

$$\begin{aligned} &R(N(v), [p_I]) \\ &\quad = R(N(v_1), [p_I^{(1)}]) \times \cdots \times R(N(v_n), [p_I^{(n)}]) \cup \{[p_I], [p_O]\} \\ &|R(N(v), [p_I])| \\ &\quad = |R(N(v_1), [p_I^{(1)}])| \times \cdots \times |R(N(v_n), [p_I^{(n)}])| + 2 \\ &s(v) = \prod_{i=1}^{n} s(v_i) + 2 = \text{Eq. (4.4)} \end{aligned}$$

Q.E.D.

Based on Lemma 4.4, we propose a polynomial time algorithm to solve the problem for the PTB WF-nets. To calculate the number of all possible states in a PTB WF-net, the proposed algorithm utilizes its process tree. The proposed algorithm is based on Depth-First Search (DFS) [55]. The tree traversal is in post-order. Let v be the most recently finished node[1] in the DFS, $s(v)$ is the number of state at v which is calculated. We propose the algorithm as follows:

≪State Number Calculation of PTB WF-net≫

Input: Process tree π of PTB WF-net $(N, [p_I])$
Output: State number $|R(N, [p_I])|$

CalculateStateNumberPTBWF-Net($(N, [p_I]), \pi$)
1 $v \leftarrow$ the root of π
2 CalculateStateNumber(v)
3 Output $s(v)$ as $|R(N, [p_I])|$, and stop

CalculateStateNumber(v)
1 **if** v is a leaf node
2 $\quad s(v) \leftarrow 2$
3 **if** v is '$\rightarrow$'
4 $\quad$ **for each** child u of v
5 $\quad\quad$ CalculateStateNumber(u)
6 $\quad s(v) \leftarrow \sum_{\text{child } u \text{ of } v}(s(u) - 1) + 1$
7 **if** v is '$\times$'
8 $\quad$ **for each** child u of v

[1] A node is said to be finished if all of its children nodes have been explored.

9 CalculateStateNumber(u)
10 $s(v) \leftarrow \sum_{\text{child } u \text{ of } v}(s(u) - 2) + 2$
11 **if** v is '$\wedge$'
12 **for each** child u of v
13 CalculateStateNumber(u)
14 $s(v) \leftarrow \prod_{\text{child } u \text{ of } v} s(u) + 2$

Theorem 4.4 *The state number calculation problem can be solved in polynomial time for PTB WF-nets with initial marking* $[p_I]$. □

Proof: Algorithm ≪State Number Calculation of PTB WF-net≫ can run in polynomial time because it is based on DFS.

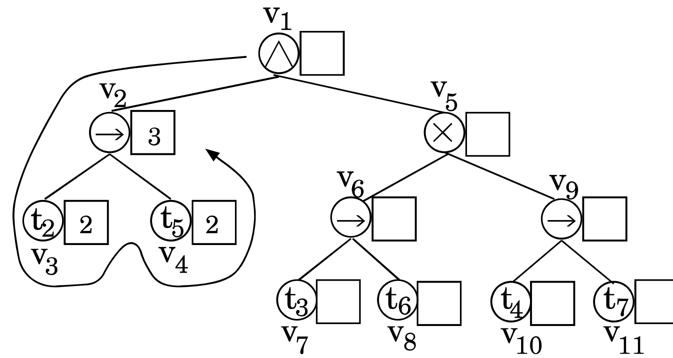
Q.E.D.

As an example, we calculate WF-net N_1 shown in Fig. 4.2 (a). The process tree is $\Pi_1{=}\wedge(\rightarrow(t_2, t_5), \times(\rightarrow(t_3, t_6), \rightarrow(t_4, t_7)))$ as shown in Fig. 4.9 (a). We apply the proposed algorithm to $(N_1, [p_1])$. Figure 4.9 shows the execution. For each node v, the rectangle of its right side represents $s(v)$. (a) The state in which v_2 was finished in the DFS. From equation $s(v) = \sum_{i=1}^{n}(s(v_i) - 1) + 1$ which v is sequence ($\rightarrow$), then we have $s(v_2) = (s(v_3){-}1) + (s(v_4){-}1) + 1 = 3$. (b) The state in which v_5 was finished. From equation $s(v) = \sum_{i=1}^{n}(s(v_i) - 2) + 2$ which v is exclusive choice ($\times$), then we have $s(v_5) = (s(v_6){-}2) + (s(v_9){-}2) + 2 = 4$. (c) The state in which v_1 was finished. From equation $s(v) = \prod_{i=1}^{n} s(v_i) + 2$ which v is parallel ($\wedge$), then we have $s(v_1) = s(v_2){\times}s(v_5) + 2 = 14$. Thus the algorithm outputs 14 as $|R(N_1, [p_1])|$.

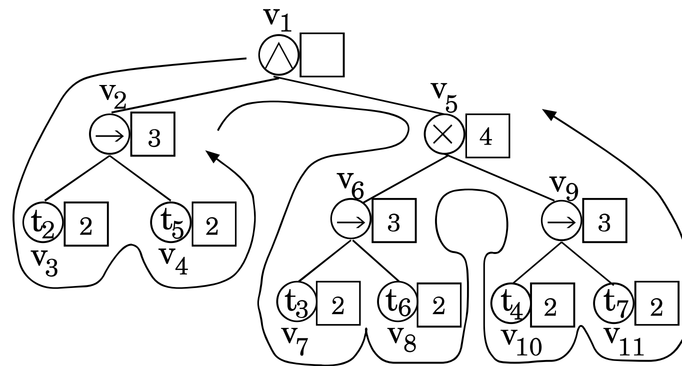
4.2.3 Application Example

Model checking is a promising method in analysis of Petri nets. A model checking tool, SPIN has been widely used in [60] and [61]. Yamaguchi et al. [60] utilized SPIN for the verification of WF-net's soundness. Hichami et al. [61] also proposed a verification method of task execution in a process chain with SPIN.

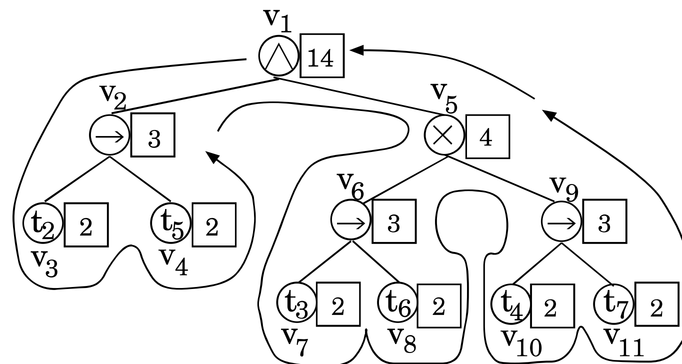
SPIN is available to a system with less than 1 million states. Thus we apply our proposed method so that we can decide whether we should use SPIN for a given WF-net. Figure 4.10 shows our proposed application in model checking. Before using SPIN, we check the state number of the input WF-net. If the state number is less than 1 million states, we can proceed to model checking with SPIN. Otherwise, we have to use other tools or split the WF-net into parts with less than 1 million states and proceed to model checking.



(a) The state in which v_2 was finished



(b) The state in which v_5 was finished



(c) The state in which v_1 was finished

Figure 4.9: The execution of the proposed algorithm for the process tree of N_1.

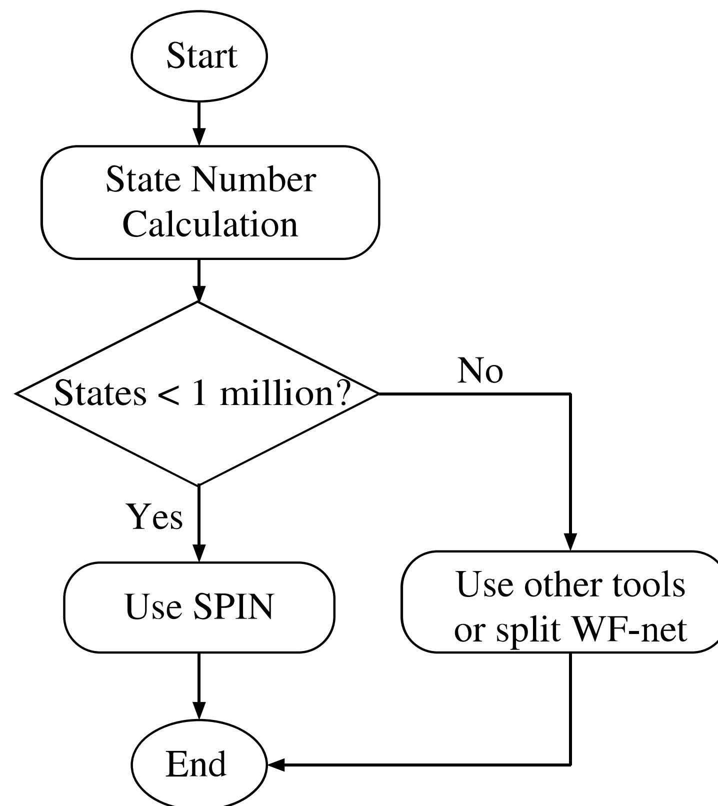


Figure 4.10: Application of state number calculation.

4.3 Behavioral Inheritance Analysis

For minor upgrade, we propose behavioral inheritance analysis. It verifies overall behavior of the program. Based on Theorem 3.1, we construct a procedure for solving the behavioral inheritance analysis problem.

≪Backward Compatibility Verification≫

Input: Original C program π, extended C program π', specification α

Output: Does π' implement specification α?

1° Translate original program π and extended program π' into WF-net N_π and $N_\pi{}'$ by utilizing ≪Program to Petri Net's Translation≫.

2° ▹ $N_\pi{}'$ Check if inherits the behavior of N_π.
Apply the backtracking algorithm [40]. If yes, proceed to Step 3°. Otherwise, output 'no' and stop.

3° Output 'yes' and stop.

The backtrack algorithm checks the partitions of reachability tree when possible. It prunes the search tree of partitioning (hidden partition and block partition) when possible. The algorithm runs a fast check on those partitions based on necessary conditions that have to hold when the WF-nets and their subnets are branching bisimilar. If the fast check fails, then they are not branching bisimilar. As a result, the backtracking algorithm can prune the entire subtree non-exhaustively. We can also also apply a notion called as protocol inheritance [62]. The method can be done only by checking the structure known as handles in the Petri nets but is restricted to acyclic extended free-choice WF-nets.

Our method is available for a wide class of Petri nets that can represent structured programs including concurrent programs because Petri net can represents concurrent process.

As an example, let us assume there are program X and program Y. Let us consider the behavioral inheritance of both programs. This can be formalized as Instance 6.

Instance 1

Instance : WF-net N_X (See Fig. 4.11(a)), WF-net N_Y (See Fig. 4.11(b))
Question : Does N_Y inherit the behavior of N_X? □

We apply ≪Backward Compatibility Verification≫ to program X and program Y. In Step 1°, we first translate program X and Y into WF-net models N_X and N_Y. The converted models are

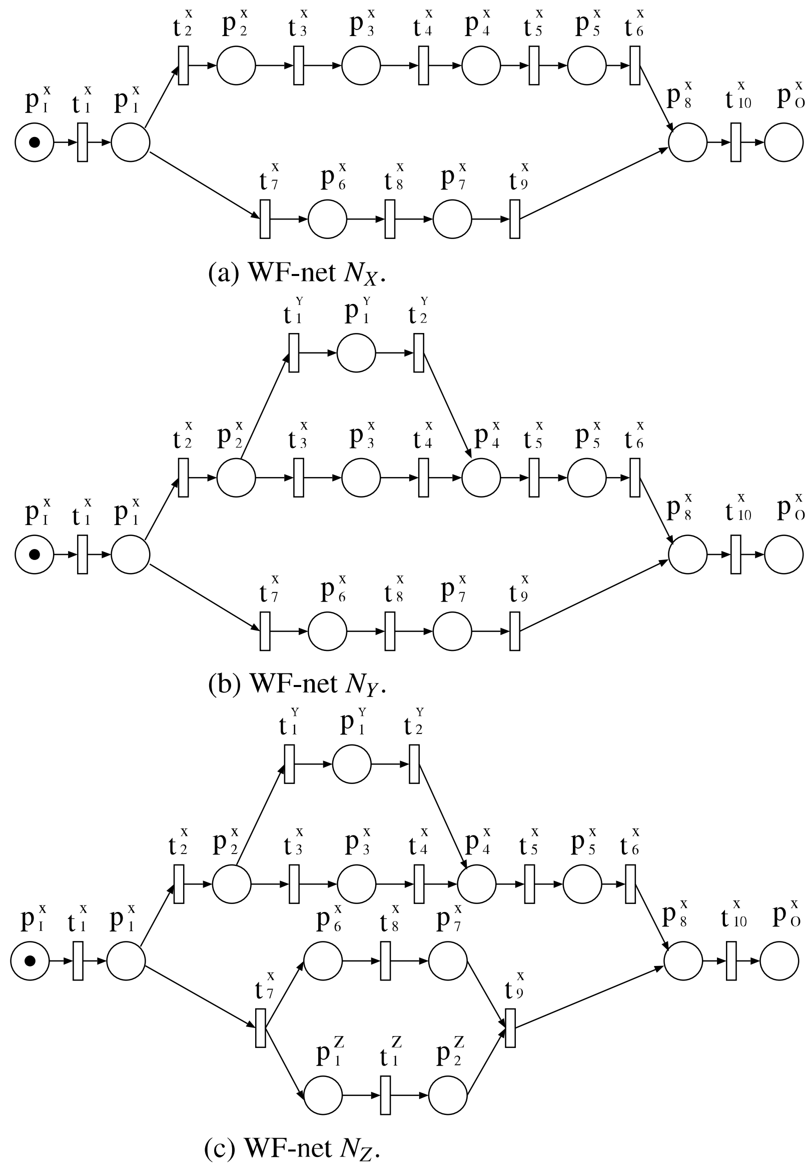


Figure 4.11: WF-nets N_X, N_Y and N_Z. N_Y derived from N_X and N_Z derived from N_Y.

shown in Fig. 4.11(a) and (b). Next, in Step 2° we check if N_Y inherits the behavior of N_X. We apply the backtracking algorithm in Step 2-2° to check if N_Y is a subclass of N_X under life-cycle inheritance. The algorithm can traverse the reachability tree with depth-first search. Figure 4.12 shows the comparison of reachability tree $R(N_X, [p_I^X])$ and $R(N_Y, [p_I^X])$ of N_X and N_Y. The dotted lines represent the firing sequences of the markings in the reachability tree which was added in N_Y after the evolution. $R(N_Y, [p_I^X])$ contains all markings and transition sequences which exist in $R(N_X, [p_I^X])$. We obtained that N_Y inherits the behavior of N_X because the states in N_X was preserved in N_Y. In Step 3°, the procedure outputs yes. Therefore based on Theorem 1, N_Y is a

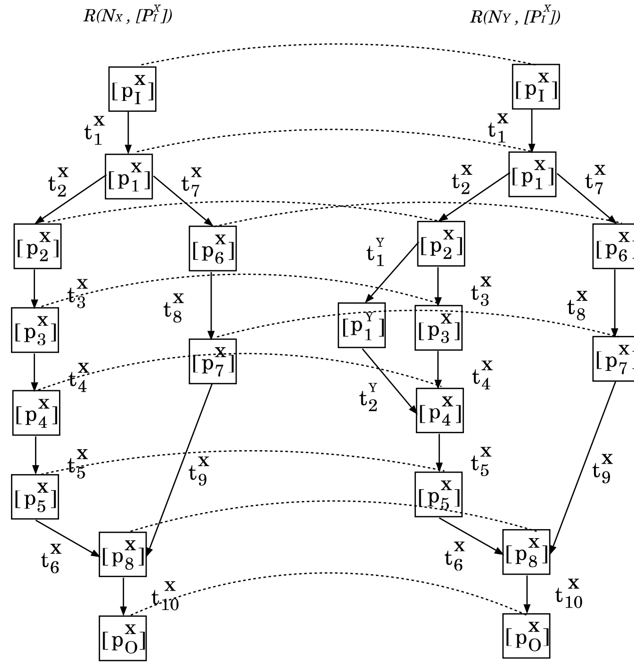


Figure 4.12: The comparison of reachability tree $R(N_X, [p_I^X])$ and $R(N_Y, [p_I^X])$ of N_X and N_Y. $R(N_Y, [p_I^X])$ is branching bisimilar to $R(N_X, [p_I^X])$ because $R(N_Y, [p_I^X])$ contains all markings that exist in $R(N_X, [p_I^X])$.

subclass of N_X under life-cycle inheritance. Therefore, N_Y inherits the behavior of N_X.

Next, let us consider a new program Z where Z derived from Y. So we consider Instance 2.

Instance 2

Instance : WF-net N_Y (See Fig. 4.11(b)), WF-net N_Z (See Fig. 4.11(c))

Question : Does N_Z inherit the behavior of N_Y ? □

Similarly, we apply the procedure ≪Security Protocol Compatibility Verification≫. Figure 4.13 shows the comparison of reachability tree $R(N_Y, [p_I^X])$ and $R(N_Z, [p_I^X])$ of N_Y and N_Z. There exist $[p_1^X][N_Y, t_7^X\rangle[p_6^X]$ and $[p_6^X][N_Y, t_8^X\rangle[p_7^X]$ in $R(N_Y, [p_I^X])$ but not in $R(N_Z, [p_I^X])$. The procedure outputs no because N_Z is not a subclass of N_Y under life-cycle inheritance. Therefore N_Z does not inherit the behavior of N_Y.

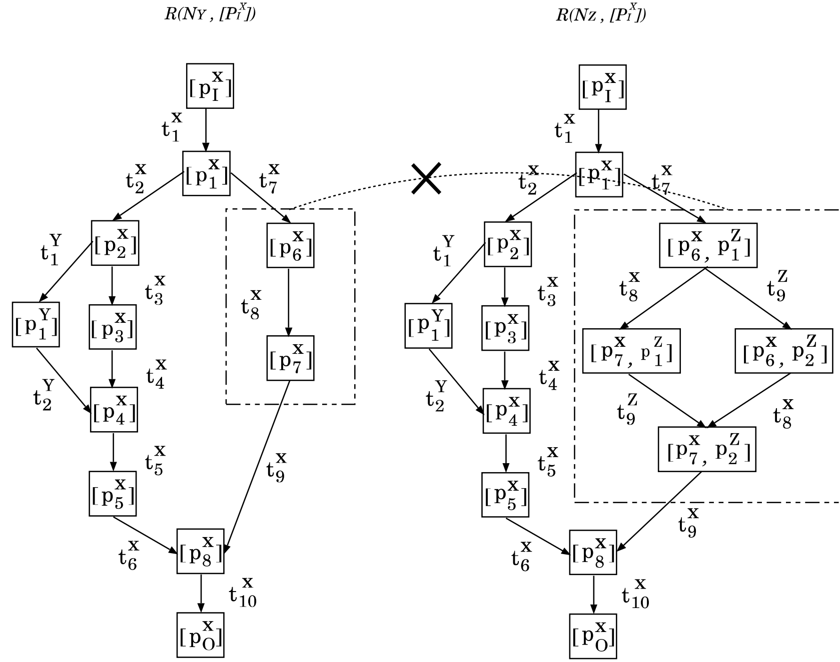


Figure 4.13: The comparison of reachability tree $R(N_Y, [p_I^X])$ and $R(N_Z, [p_I^X])$ of N_Y and N_Z. There exist $[p_1^X][N_Y, t_7^X\rangle[p_6^X]$ and $[p_6^X][N_Y, t_8^X\rangle[p_7^X]$ in $R(N_Y, [p_I^X])$ but not in $R(N_Z, [p_I^X])$.

4.4 Response Property Analysis

For major upgrade, we propose response property analysis. It verifies statement execution sequence. Response property [23] is a kind of liveness property that, given two transition α and β, "whenever α is executed, then β has to be eventually executed afterwards". We can generalize the problem as if $\alpha_1, \alpha_2, \cdots, \alpha_k$ fire, then does $\beta_1, \beta_2, \cdots, \beta_m$ always fire? We restricted the analysis of response property to acyclic WF-net. This restriction is reasonable because acyclic WF-nets are applicable to analyze most actual systems.

Let $\mathcal{L}(N, M_0) \subseteq L(N, M_0)$ be the set of any firing sequence that transforms $[p_I]$ to any dead marking. Note that in any acyclic WF-net N=(P, T, A), any transition can fire at most once in a firing sequence and every marking in $R(N, [p_I])$ eventually reaches a dead marking. For transitions α and β in any firing sequence, $\alpha \prec \beta$ denotes that α precedes β. We assume that every transition can fire only once. We formalize response property as follows:

Definition 4.5 (Response Property) *For an acyclic WF-net N=(P, T, A), a subset $T_\beta \subseteq T$ is said to respond to a subset $T_\alpha \subseteq T$ if $\forall(\alpha,\beta) \in T_\alpha \times T_\beta, \forall\sigma \in \mathcal{L}(N, [p_I]) : \alpha \in \sigma \Rightarrow \beta \in \sigma, \alpha \prec \beta$.* □

We can define a problem of deciding this property as follows:

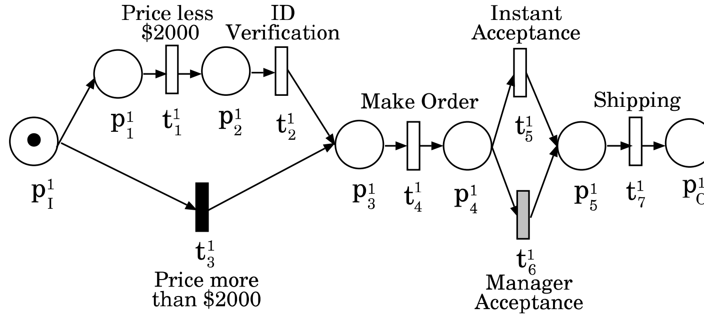


Figure 4.14: An example workflow N_1 of an online ordering service.

Definition 4.6 (Response Property Problem)
Instance: WF-net N, transitions $\alpha_1, \alpha_2, \cdots, \alpha_n, \beta_1, \beta_2, \cdots, \beta_m$ *of N.*
Question: Do $\beta_1, \beta_2, \cdots$ *, and* β_m *respond to* $\alpha_1, \alpha_2, \cdots$ *, and* α_n*?* □

Let us consider three instances of response property problem as examples. The instances is shown in Fig. 4.14, Fig. 4.15 and Fig. 4.16. Instance in Fig. 4.14 has been considered in Sect. 1. We concretely discuss the reason in this chapter.

Instance 3
Instance: WF-net $(N_1, [p_I])$*, transitions* t^1_6 *and* t^1_3 *(See Fig. 4.14).*
Question: Does t^1_6 *respond to* t^1_3 *?* □

The answer for Instance 3 is no, because t^1_6 does not always fire for $[p^1_I][\sigma, N_1\rangle[p^1_5]$ because there exists firing sequences σ=$t^1_1t^1_2t^1_4t^1_5$ or σ=$t^1_3t^1_4t^1_5$ transforming $[p^1_I]$ to $[p^1_5]$. Then t^1_6 does not respond to t^1_1. This means that Petri net structure plays an important role to the response property as discussed in Sect. 1.

Instance 4
Instance: WF-net $(N_2, [p^2_I])$*, transitions* t^2_1 *and* t^2_4 *(See Fig. 4.15).*
Question: Does t^2_4 *respond to* t^2_1*?* □

The answer for Instance 4 is yes, because t^2_4 always fires for $[p^2_I][\sigma, N_2\rangle[p^2_3, p^2_5, p^2_7]$. There are firing sequences such as σ=$t^2_1t^2_2t^2_3t^2_{10}t^2_4$, σ=$t^2_1t^2_{10}t^2_2t^2_3t^2_4$ and σ=$t^2_1t^2_2t^2_{10}t^2_3t^2_4$ where t^2_4 always exists in σ. Then t^2_4 responds to t^2_1. This is because the sub-marking $[p^2_3, p^2_5]$ that enables t^2_4 is reachable from $[p^2_I]$. Hence, the reachability of marking is important to the analysis of response property.

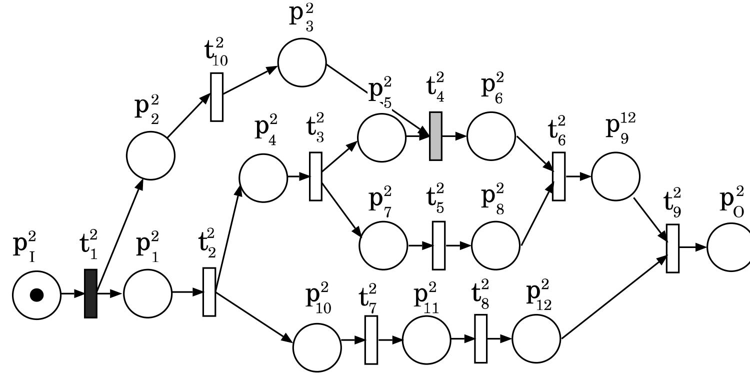


Figure 4.15: Instance 2: A WF-net N_2.

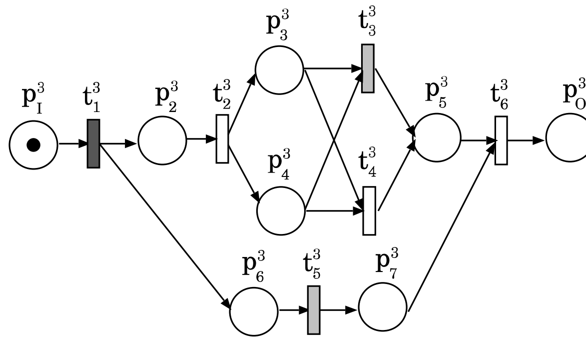


Figure 4.16: Instance 3: WF-net N_3.

Instance 5

Instance: WF-net $(N_3, [p_I^3])$*, transitions* t_1^3*,* t_3^3 *and* t_5^3 *(See Fig. 4.16).*
Question: Do t_3^3 *and* t_5^3 *respond to* t_1^3 *?* □

The answer for Instance 5 is no, because t_3^3 and t_5^3 only fire for $[p_I^3][\sigma, N_2\rangle[p_3^3, p_4^3, p_6^3]$ when there are firing sequences $\sigma{=}t_1^3t_2^3t_3^3t_5^3$, $\sigma{=}t_1^3t_2^3t_5^3t_3^3$ and $\sigma{=}t_1^3t_5^3t_2^3t_3^3$. However, if t_4^3 fires then t_3^3 will not fires. Then t_3^3 and t_5^3 do not respond to t_1^3. Note that although the marking $[p_3^3, p_4^3, p_6^3]$ that enables t_3^3 and t_5^3 is reachable from $[p_I^3]$, it does not guarantee that t_3^3 and t_5^3 will fire. We also need to consider the complexity of this problem.

From the analysis result of Instances 3–5, some markings which enable the firing of the transitions in the response property may be reachable but does not always guarantee the firing of those transitions. Hence, we can conclude that in liveness property i.e L1-liveness [32], a particular transition must fire at least once from any reachable markings, but in response property, β must be guaranteed from the marking where α was fired.

4.4.1 Decidability and Complexity of Response Property

We tackle the response property problem restricted to n=m=1, named simple response property problem. Let us consider the decidability of the simple response property problem.

Theorem 4.5 *The simple response property problem is decidable for sound WF-nets.* □

Proof : Let (N, M_0) be a sound WF-nets. Since the given WF-net N is sound, all markings are reachable to $[p_O]$. If all of the firing sequence to $[p_O]$ from a marking after α is immediately executed contains β, then β responds to α. Therefore, we only need to check whether the firing sequence to $[p_O]$ from a marking after α is immediately executed contains β or not. Because $(N, [p_I])$ is bounded, we can construct the reachability tree. By using the reachability tree, it is possible to clearly determine whether β exists or not. **Q.E.D.**

Let us consider the computation complexity of the simple response property problem. We call the problem as intractable if it is not solvable in polynomial time.

To do so, we show that an NP-complete problem, called 3-conjunctive normal form boolean satisfiability problem (3-CNF-SAT for short), can be transformed to the simple response property problem of AC WF-nets.

Definition 4.7 (3-CNF-SAT)
Instance: Expression $\mathcal{E}$ of 3-conjunctive normal form that has n boolean variables and m clauses.
Question: Is there an assignment of variables satisfying $\mathcal{E}$=true? □

Let us consider an AC WF-nets shown in Fig. 4.17. We need to check does β respond to α? The constructed WF-nets shows that if $\mathcal{E}$=*true* then marking $[p_2, p_4]$ is reachable from $[p_I]$. However, β will not always fires because of the conflict with γ. In this case, β does not respond to α. We give the following theorem on the complexity:

Theorem 4.6 The simple response property problem is co-NP hard for acyclic AC WF-nets. □

Proof: We prove the NP-hardness by a reduction from 3-CNF-SAT. Let $\mathcal{E}$ be an expression of 3-CNF-SAT which has n boolean variables $x_1, x_2, \cdots, x_n$ and m clauses $c_1, c_2, \cdots, c_m$. A literal ℓ_i is either a variable x_i or its negation $\overline{x_i}$. Without loss of generality, it can be assumed that $\mathcal{E}$ has all of x_i's and $\overline{x_i}$'s (i=$1, 2, \cdots, n$), and m≥3.

We construct a WF-net $N_\mathcal{E}$=$(P_\mathcal{E}, T_\mathcal{E}, A_\mathcal{E})$ with two transitions α and β, and show that $\mathcal{E}$ is satisfiable iff β does not respond to α in $(N_\mathcal{E}, [p_I])$. $N_\mathcal{E}$=$(P_\mathcal{E}, T_\mathcal{E}, A_\mathcal{E})$ is given as follows.

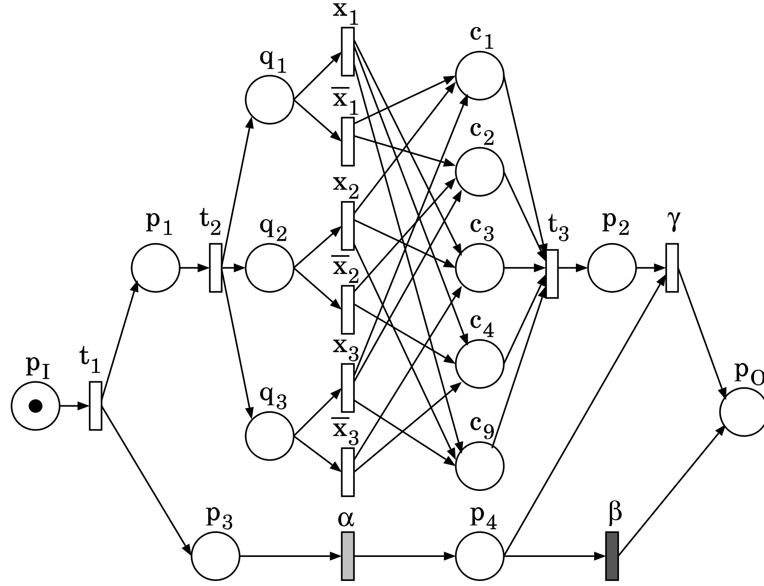


Figure 4.17: The AC WF-net $(N_{\mathcal{E}_1}, [p_1])$ corresponding to a 3-CNF-SAT expression $\mathcal{E}_1 = (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3)$. $\mathcal{E}_1$ is satisfiable. β does not respond to α in $(N_{\mathcal{E}}, [p_I])$

$$
\begin{aligned}
P_{\mathcal{E}} = \ & \{p_I, p_1, p_2, p_3, p_4, p_O\} \cup \textstyle\bigcup_{i=1}^{n}\{q_i\} \cup \bigcup_{j=1}^{m}\{c_j\} \\
T_{\mathcal{E}} = \ & \{t_1, t_2, t_3, \alpha, \beta, \gamma\} \cup \textstyle\bigcup_{i=1}^{n}\{x_i, \overline{x_i}\} \\
A_{\mathcal{E}} = \ & \{(p_I, t_1), (t_1, p_1), (p_1, t_2), (t_3, p_2), (p_2, \gamma), (t_1, p_3), \\
& (p_3, \alpha), (\alpha, p_4), (p_4, \gamma), (\gamma, p_O), (p_4, \beta), (\beta, p_O)\} \\
& \cup \textstyle\bigcup_{i=1}^{n}\{(t_2, q_i), (q_i, x_i), (q_i, \overline{x_i})\} \\
& \cup \textstyle\bigcup_{k=1}^{3} \bigcup_{j=1}^{m}\{(\ell_k, c_j) | \ell_k \text{ is the } k\text{-th literal of clause } c_j\} \\
& \cup \textstyle\bigcup_{j=1}^{m}\{(c_j, t_3)\}
\end{aligned}
$$

$N_{\mathcal{E}_1}$ is an AC WF-net because its short-circuited net $\overline{N_{\mathcal{E}}}$ is strongly connected; Places p_2 and p_4 share an output transition γ while p_4 has another output transition β; Places $c_1, c_2, \cdots, c_m$ share only one output transition t_2, and the other places share no output transition. $N_{\mathcal{E}}$ can be constructed in polynomial time, because it consists of $(n{+}m{+}6)$ places, $(2n{+}5)$ transitions, and $(3n{+}4m{+}12)$ arcs.

The proof of "if" part: Let μ denote an assignment of variables satisfying $\mathcal{E}{=}\mathit{true}$, and let $\ell_1, \ell_2, \cdots, \ell_n$ be the literals mapped to *true* by μ. By the construction of $N_{\mathcal{E}_1}$, we have

$$
\begin{aligned}
[p_1] \quad & [N_{\mathcal{E}}, t_1 t_2\rangle\, [q_1, q_2, \cdots, q_n, p_3] \\
& [N_{\mathcal{E}}, \ell_1 \ell_2 \cdots \ell_n\rangle\, M.
\end{aligned}
$$

Since μ satisfies $\mathcal{E}$, for each clause c_j ($1 \leq j \leq m$), there exists a literal ℓ_i ($1 \leq i \leq n$) in c_j. Therefore place c_j is marked by firing ℓ_i i.e. $M \geqq [c_1, c_2, \cdots, c_m, p_3]$. Let $M' = M - [c_1, c_2, \cdots, c_m, p_3]$.

$$
\begin{aligned}
M &= M' \cup [c_1, c_2, \cdots, c_m, p_3] \\
&[N_{\mathcal{E}}, t_3\rangle M' \cup [p_2, p_3] \\
&[N_{\mathcal{E}}, \alpha\gamma\rangle M' \cup [p_O]
\end{aligned}
$$

β is dead in $(N_{\mathcal{E}}, M' \cup [p_O])$. Thus, β does not respond to α.

The proof of "only if" part: Let μ denote any assignment of variables satisfying $\mathcal{E}$=*false*. Since μ does not satisfy $\mathcal{E}$, there exists a clause c_j ($\in \{c_1, c_2, \cdots, c_m\}$) mapped to *false* by μ. Let $\ell_1^j, \ell_2^j, \ell_3^j$ denote the literals in c_j. Since the corresponding transitions $\ell_1^j, \ell_2^j, \ell_3^j$ do not fire, their common output place, i.e. place c_j, is never marked. c_j is an input place of transition t_3, so t_3 is dead. We have

$$
\begin{aligned}
[p_I] \quad & [N_{\mathcal{E}}, t_1\rangle\, [p_1, p_3] \\
& [N_{\mathcal{E}}, *\rangle\, [p_3] \cup M \qquad (\forall M \in R(N_{\mathcal{E}}, [p_1])) \\
& [N_{\mathcal{E}}, \alpha\rangle\, [p_4] \cup M \\
& [N_{\mathcal{E}}, *\rangle\, [p_4] \cup M' \qquad (\forall M' \in R(N_{\mathcal{E}}, M)) \\
& [N_{\mathcal{E}}, \beta\rangle\, [p_O] \cup M'
\end{aligned}
$$

Thus β responds to α. **Q.E.D.**

For example, let us consider the following boolean expression:

$$
\begin{aligned}
\mathcal{E}_1 = {} & (\overline{x_1} \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \\
& \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3)
\end{aligned}
$$

$\mathcal{E}_1$ is satisfiable by choosing x_1=*true*, x_2=*true*, x_3=*true*. Figure 4.17 shows the Petri net $N_{\mathcal{E}_1}$ constructed from $\mathcal{E}_1$. β does not respond to α, because

$$
\begin{aligned}
[p_I] \quad & [N_{\mathcal{E}_1}, t_1t_2\rangle\, [p_1, p_3] \\
& [N_{\mathcal{E}_1}, x_1x_2x_3\rangle\, [{c_1}^2, {c_2}^1, {c_3}^2, c_4, {c_5}^3, p_3] \\
& [N_{\mathcal{E}_1}, t_3\rangle\, [c_1, c_3, {c_5}^2, p_2, p_3] \\
& [N_{\mathcal{E}_1}, \alpha\rangle\, [c_1, c_3, {c_5}^2, p_2, p_4] \\
& [N_{\mathcal{E}_1}, \gamma\rangle\, [c_1, c_3, {c_5}^2, p_O].
\end{aligned}
$$

From here we can say that the decision problem related to this problem, i.e. the response property problem is NP-hard. This means that the original problem is intractable for the class of acyclic AC WF-nets.

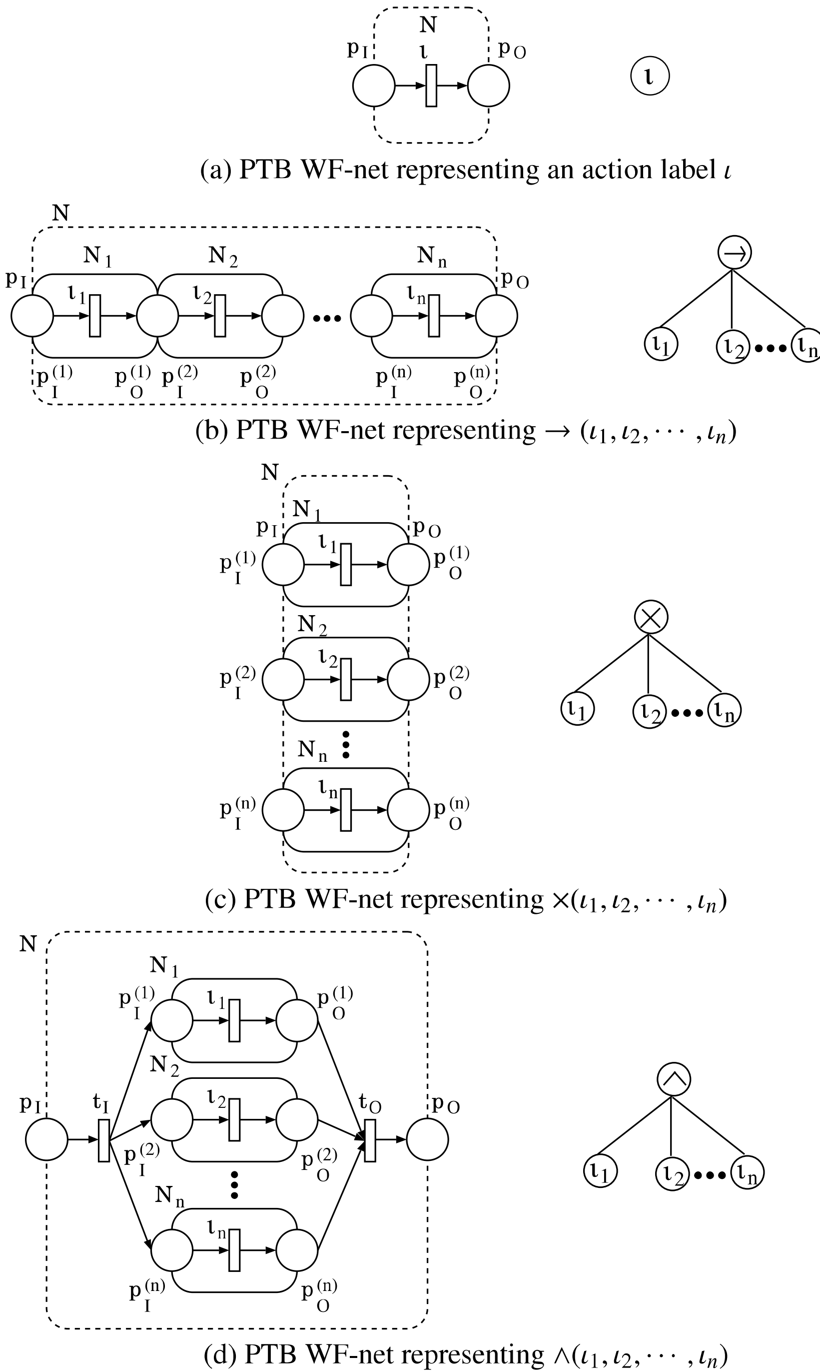
4.4.2 Polynomial Time Procedure

In this section, we propose a polynomial time procedure to decide simple response property. From Theorems 1 and 2, the response property problem is decidable but is intractable even if it is simple. So we need a solution which can be solved in polynomial time. We utilize a representational bias of Petri net called as process tree to solve the simple response property problem. Process tree can be used to tackle analysis problem regarding to state space problem [63, 25]. Compared to the abstraction method proposed in Ref. [61], process tree preserves accurate representation of the WF-net structure.

First, let us consider the simplest form of process tree with only one parent node and n child nodes.

Lemma 4.5 *For a given PTB WF-net N=(P, T, A), let Π_N be its process tree which has the parent node of α and β as a sequence operator ($\rightarrow$). β responds to α if α is on the left of β in Π_N, i.e for α=$t^{(i)}$ and β=$t^{(j)}$, $i < j$ holds.* □

Proof: If $\oplus$ is $\rightarrow$, then N is a WF-net constructed by concatenating subnet $N_1, N_2, \cdots, N_n$ which connect the sink place $p_O^{(i)}$ of N_i with the source place $p_I^{(k+1)}$ of $N_{k+1}(1 \leq k < n)$ (See Fig. 4.18(a)). In N_n, $[p_I]$ (=$[p_I^{(1)}]$) is reachable to $[p_O^{(1)}]$, $[p_I^{(2)}]$ (=$[p_O^{(1)}]$) is reachable to $[p_O^{(2)}]$, $\cdots$, $[p_I^{(n)}]$ (=$[p_O^{(n-1)}]$) is reachable to $[p_O^{(n)}]$ (=$[p_O]$), because $N_1, N_2, \cdots, N_n$ is sound. i<j holds for α=$t^{(i)}$ and β=$t^{(j)}$ where the parent node is a sequence operator. Therefore, $\forall \sigma \in \mathcal{L}(N, [p_I])$ such that $[p_I][N, \sigma\rangle[p_O]$ and $\alpha \prec \beta$ holds. Hence, β will always fire after α. **Q.E.D.**



(a) PTB WF-net representing an action label ι

(b) PTB WF-net representing $\rightarrow(\iota_1, \iota_2, \cdots, \iota_n)$

(c) PTB WF-net representing $\times(\iota_1, \iota_2, \cdots, \iota_n)$

(d) PTB WF-net representing $\wedge(\iota_1, \iota_2, \cdots, \iota_n)$

Figure 4.18: Illustration of PTB WF-net and its equivalent process tree.

Lemma 4.6 *For a given PTB WF-net $N=(P,T,A)$, let Π_N be its process tree and $\oplus$ as the parent node of α and β. β does not respond to α if $\oplus$ is an exclusive-choice operator $\times$.* □

Proof: If $\oplus$ is $\times$, then N is a WF-net constructed by bundling $N_1, N_2, \cdots, N_n$ which forms a selective concurrent paths between their source places and sink places (See Fig. 4.18(b)). In

N_n, p_I=$p_I^{(1)}$=$p_I^{(2)}$=$\cdots$=$p_I^{(n)}$ and p_O=$p_O^{(1)}$=$p_O^{(2)}$=$\cdots$=$p_O^{(n)}$. Note that only one path will allow the transition t_n in N_n to fires. Therefore there exists different firing sequences $\sigma_1, \sigma_2, \cdots \sigma_n$ for each path in N_1, N_2, $\cdots$ N_n for $[p_I][N_n, \sigma_n\rangle[p_O]$. Hence, only either α or β will fire. **Q.E.D.**

Lemma 4.7 *For a given PTB WF-net N=(P, T, A), let Π_N be its process tree and $\oplus$ as the parent node of α and β. β does not respond to α if $\oplus$ is a parallel operator $\wedge$.* □

Proof: If $\oplus$ is $\wedge$, then N is a WF-net constructed by joining respectively all source places of the concurrent paths with a transition t_I, and sink places with a transition t_O of PTB WF-net $N_1, N_2, \cdots, N_n$ (See Fig. 4.18(c)). Since N_1, N_2, $\cdots$, and N_n are sound, $[p_I^{(1)}, p_I^{(2)}, \cdots, p_I^{(n)}]$ is reachable to $[p_O^{(1)}, p_O^{(2)}, \cdots, p_O^{(n)}]$. $p_O^{(1)}$, $p_O^{(2)}$, $\cdots$, and $p_O^{(n)}$ are connected to p_O via another additional transition t_O. Therefore there exist partial firing sequences σ=$\alpha\sigma_1\beta$ and σ=$\beta\sigma_1\alpha$ where σ_1 is all possible firing sequences in $\{\alpha,\beta\}\backslash T$ for $[p_I^{(1)}, p_I^{(2)}, \cdots, p_I^{(n)}][N_n, \sigma\rangle[p_O^{(1)}, p_O^{(2)}, \cdots, p_O^{(n)}]$. Hence, based on σ, β will not always fires after α. **Q.E.D.**

Based on Lemmas 1–3, we can conclude that sequence operator is important for response property while exclusive-choice and parallel operator does not guarantee response property. The sequence operator guarantees sequential firing of α before β. However, the given condition in Lemmas 1–3 only hold for the simplest form of a process tree which is necessary but not sufficient. Therefore, we need to consider the necessary and sufficient condition i.e when the sequence operator is not a direct parent of α and β.

Let us consider the following problem. Let υ_{NCA} be the nearest common ancestor node of α and β in a process tree Π of a WF-net N where there exists the shortest path from α via υ_{NCA} to β. Does β respond to α if υ_{NCA} is a sequence operator? We consider two case: (1) the position of α and β in the process tree; and (2) there is an exclusive-choice or a parallel operator between the path from υ_{NCA} to α or β. Case (1): If the position of α is on the left of β in Π_N, then α will fire before β in the sequence construct in N, respectively. However, if α is on the right of β, then β will fire before α. Case (2): If there is an exclusive-choice operator or parallel operator between υ_{NCA} and α, then α will fire at least once regardless of the choice and parallel construct. Therefore, we can ignore the operator between υ_{NCA} and α. In case of the operator between υ_{NCA} and β, it is clear that in parallel construct, β will always fire. However, there is a case that β will not fire in a choice construct. We can give the following theorem:

Theorem 4.7 *For a given PTB WF-net N=(P, T, A) with transitions α and β. β responds to α iff in the process tree Π_N*

(i) The nearest common ancestor v_{NCA} of α and β is a sequence operator ($\rightarrow$); and

(ii) The position of α is on the left of β, i.e for α=$t^{(i)}$ and β=$t^{(j)}$, $i < j$ holds; and

(iii) There is no exclusive-choice operator ($\times$) between the path from v_{NCA} to β □

Proof: The proof of “if” part: Let us consider two cases in Π_N: (1) v_{NCA} is the parent node of α and β; and (2) v_{NCA} is a non-parent node. Case (1): If v_{NCA} is the parent node of α and β, this case corresponds to the precondition of Lemma 1. Case (2): Condition (i), (ii) and (iii) can be illustrated by the process tree Π_N shown in Fig. 4.19. $\oplus_\alpha$ denotes the ancestor nodes of α and the descendants of v_{NCA}. $\oplus_\beta$ is defined similarly as $\oplus_\alpha$. Π_N can be illustrated with the WF-net N shown in Fig. 4.20. N consists of N_A, N_B and N_C. If v_{NCA} is a non-parent node, then α and β is in the different connected subnets in N. α is in N_A and β is in N_B and N_C connects N_A and N_B. t_I (of N_B) can always fires after α and that β can always be enabled after t_I fires since N_B has no choice.

The proof of “only-if” part: We show the proof using a process tree Π_N which has a subtree π_ϕ with root v_{NCA} where α and β are the leaf nodes.

In Condition (i), if v_{NCA} is not sequence ($\rightarrow$), then v_{NCA} is either an exclusive-choice operator $\times$ or parallel operator $\wedge$. We consider two cases: (1) v_{NCA} is the parent node of α and β; and (2) v_{NCA} is a non-parent node. Case (1): If v_{NCA} is the parent node of α and β, the proof is immediate from Lemmas 2 and 3. Case (2): If v_{NCA} is a non-parent node as illustrated in Fig. 4.19, there exists a path from v_{NCA} via $\oplus_\alpha$ and $\oplus_\beta$ to α and β. We only need to confirm the firing of β only when α fires. Nevertheless, α will fire at least once with any operator. Therefore, we can ignore the operator of $\oplus_\alpha$. We only need to consider when $\oplus_\beta$=$\times$ and $\oplus_\beta$=$\wedge$. If $\oplus_\beta$=$\times$, similar to Lemma 2, the firing is selective where only either transition in the subtree π_b or π_β will fire. If $\oplus_\beta$=$\wedge$, similar to Lemma 3, there exist partial firing sequences $\alpha\sigma\beta$ and $\beta\sigma\alpha$ where there is a case when β can fires before α.

In Condition (ii), if α is on the right of β then $i > j$. Therefore, there exists a case where β fires before α in N. N is acyclic, therefore the firing of each transitions is only once where there exists no firing sequence that satisfies $\alpha \prec \beta$ once β fires before α.

In Condition (iii), if there exists an exclusive-choice operator between the path from v_{NCA} to β such that $\oplus_\beta$=$\times$, then β is in an exclusive-choice construct. Therefore, there exists a case where β will not fire. **Q.E.D.**

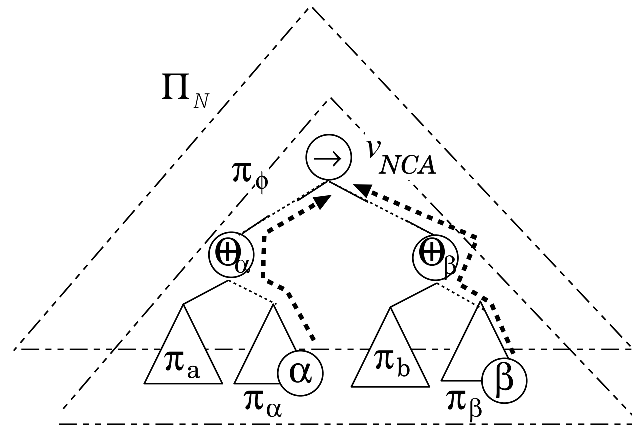


Figure 4.19: Illustration of the proof of "only-if part" of Theorem 4.7. Theorem 4.7: In a subtree π_ϕ of Π_N, β responds to α iff (i) there are paths that lead to common ancestor node $\upsilon_{NCA}= \rightarrow$ (dotted arrows); (ii) for $\alpha=t^{(i)}$ and $\beta=t^{(j)}$, $i < j$ holds; and (iii) $\oplus_\beta$ is not $\times$ (while $\oplus_\alpha$ can be any operator).

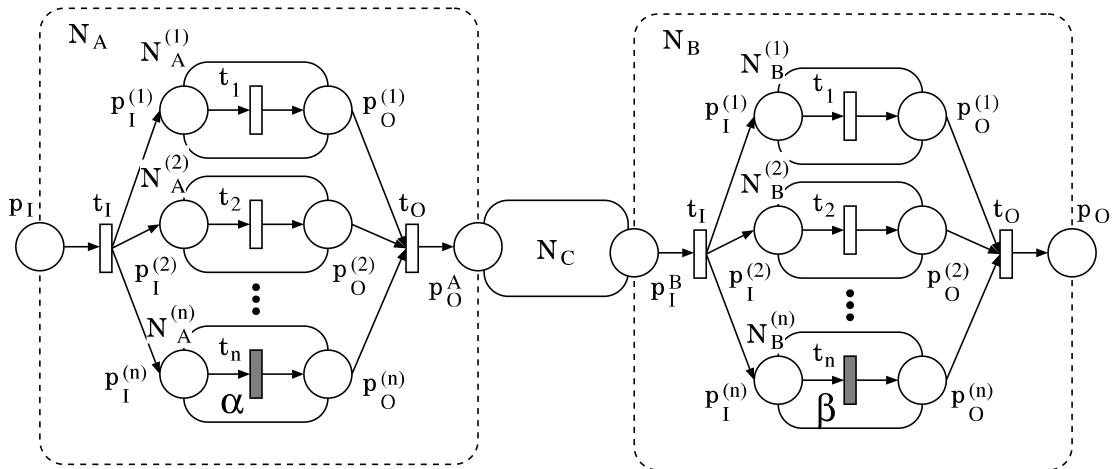


Figure 4.20: Illustration of the proof of "if part" of Theorem 4.7.

Theorem 4.7 implies the condition to decide the response property problem. So we construct a polynomial time procedure based on Theorem 4.7. The process tree can be traversed with Breadth-first Search (BFS) [57]. We give the procedure as follows:

≪Decision of Simple Response Property≫

Input: PTB WF-net N (=(P, T, A)), transitions $\alpha, \beta \in T$.

Output: Does β respond to α?

1° Convert N into process tree Π with ≪Process Tree Conversion Algorithm≫.

2° Check Conditions (i), (ii) and (iii) of Theorem 4.7.

2-1° ▹ Check Condition (i). Check if nearest common node v_{NCA} of α and β is a sequence ($\rightarrow$) from the root of Π_N.
Let the ρ_α be the path from root to α and ρ_β be the path from root to β. Let ρ_C be the common part of ρ_α and ρ_β. If the last node of ρ_C is not sequence ($\rightarrow$), then output no and stop.

2-2° ▹ Check Condition (ii).
Let i be the position of α and j be the position of β. If $i > j$, then output no and stop.

2-3° ▹ Check Condition (iii).
Backtrack from β to v_{NCA}. If a visited node is exclusive-choice ($\times$), then output no and stop.

3° Output yes and stop.

Property 4.2 *The following problem can be solved in polynomial time: Given a PTB WF-net N with two transitions α and β, to decide whether β responds to α.* □

Proof : The ≪Process Tree Conversion Algorithm≫ takes $O(|P| + |T|)$ and response property check takes $O(|T|)$ based on BFS. **Q.E.D.**

4.4.3 Application Example

In this section, we show an application example of our model-driven approach for software evolution. We assume that we need to develop a new software based on a legacy software of the i-Refrigerator. Some legacy software do not comply to their original design model. Thus, to improve the software we need to regenerate the design based on its source code. The reason to improve existing legacy software is that the users of the software are already familiar with the interface and functions but still need to improve the performance of the software or fix it with important security upgrades.

The program in Fig. 3.6 can be translated into Petri net with C2PNML by applying ≪Program to Petri Net Translation≫. The obtained Petri net is shown in Fig. 4.21. N_{π_3} represents the control flow of program π_3. Then, we implement our model-driven approach to develop a multi-threaded program for *updateDB()* and at the same time preserve the same features and usability of the program. This multi-threaded program is a requirement to improve the performance of the *updateDB()* function.

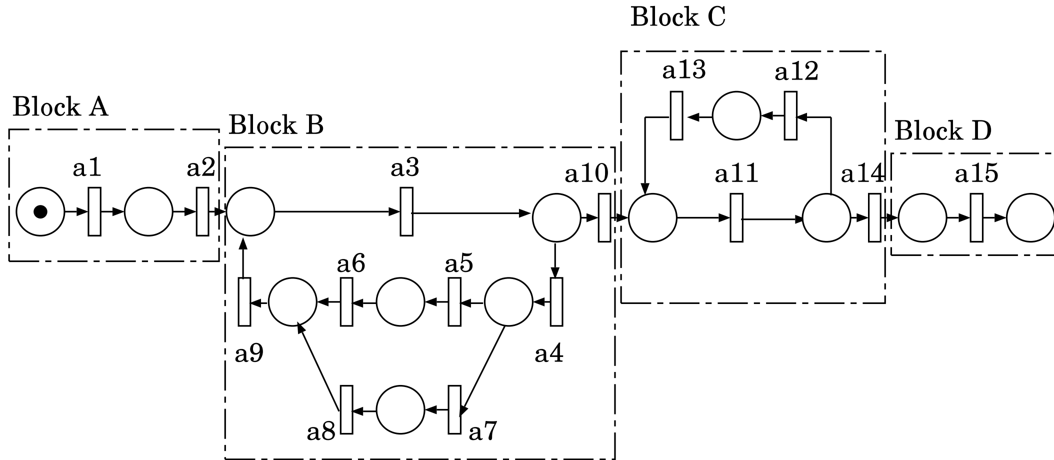


Figure 4.21: Petri net model N_{π_3}.

Based on the Petri net N_{π_3} representing program π_3, the program can be divided into 4 parts which are *Block* A, B, C and D. *Block B* and C contain the main process of the program which includes two `for` blocks. These blocks run in sequence. To increase the performance of the program, *Block B* and C can be rearranged to run in parallel. The program is rewritten as program π_4 shown in Fig. 4.22. As the result, we can obtain the model N_{π_4} as shown in Fig. 4.23. The parallel process can be implemented with `fork()` statement. Statements `a3`, `a13`, `a19`, `a20` and `a21` are important for multi-thread process in the parallel structure.

In the verification procedure, Σ_1 and Σ_2 must be satisfied. The reason why Σ_1 of π_4 must be satisfied because after thread execution of *Block B* in the child process, the execution must exits properly with statement `a13` of `_exit(status)`. Then, Σ_2 must be satisfied in program π_4 on line 5 and line 23 is that after the statement `a2` of `scanf ("%d",&stock);` executes, statement `a17` of `setID("XC-00",i);` always executes. The statements must executed according to its original sequence as in program π_3. In π_3, on line 5 and line 23 is that after the statement `a2` of `scanf ("%d",&stock);` executes, statement `a13` of `setID("XC-00",i);` always executes. We need to confirm whether the response property of π_4 complies with π_3.

In this major upgrade, we need to check the specification shown in Fig. 4.24.

As an example, let us check if π_4 satisfies response properties Σ_1. We applies ≪Decision of Response Property≫. In Step 1°, we check if N_{Π_4} is a PTB WF-net. In Step 2°, the Petri net model N_{Π_4} can be converted into process tree Π_4. In Step 3°, we check Condition (i), (ii) and (iii) by traversing Π_4 with breadth-first search. We found that the backtrack path between a_1 and a_{13} is *path*$=\rightarrow, \wedge, \rightarrow$. There is no $\times$ in *path*. As the result, in Step 4°, the procedure outputs yes. We obtained that π_4 satisfies response properties Σ_1. See Fig. 4.25 for an illustration of the

```
1 void updateDB(){
2    int stock = RetrieveSQLData(); \*a1*\
3    if ((pid = fork()) == 0){
4       // child process
5       for (i=1;i<=stock/2;i++)
6       {
7         if(i%2==0){
8          setID"ID: XA-00%d\n",i); \*a2*\
9         type = meat';
10        }else{
11 setID("ID: XB-00%d\n",i); \*a10*\
12 type = packed-food';
13   }
14 usleep(300000);
15        }
16      _exit(status); \*a13*\
17    }else if (pid > 0){
18        // parent process
19 usleep(150000);
20        for (j=(stock/2)+1;j<=stock;j++){
21            setID("ID:XC--00%d \n",j);
22             type = vegetable';
23              usleep(300000);
24        }
25 // wait for all child process
26        while(kill(pid,0));
27 wait(&status);
28    }
29  return 0;
30 }
```

Figure 4.22: A C program π_4 for extended i-Refrigerator.

procedure. We can do the same for Σ_2.

We also tested the approach to some programs that implement popular algorithms such as backpropagation algorithm in neural network, dynamic programming such as knapsack problem and sorting program. The program with the most statements has around 1,200 statements and when translated to Petri net model will have 1,400 nodes. In our verification procedure, the process tree of the Petri net model has lesser nodes which is around 670 nodes. Moreover, our model-driven procedure for the execution sequence verification utilizes breadth-first algorithm to the process tree which has only computational cost of $O(|V| + |E|)$.

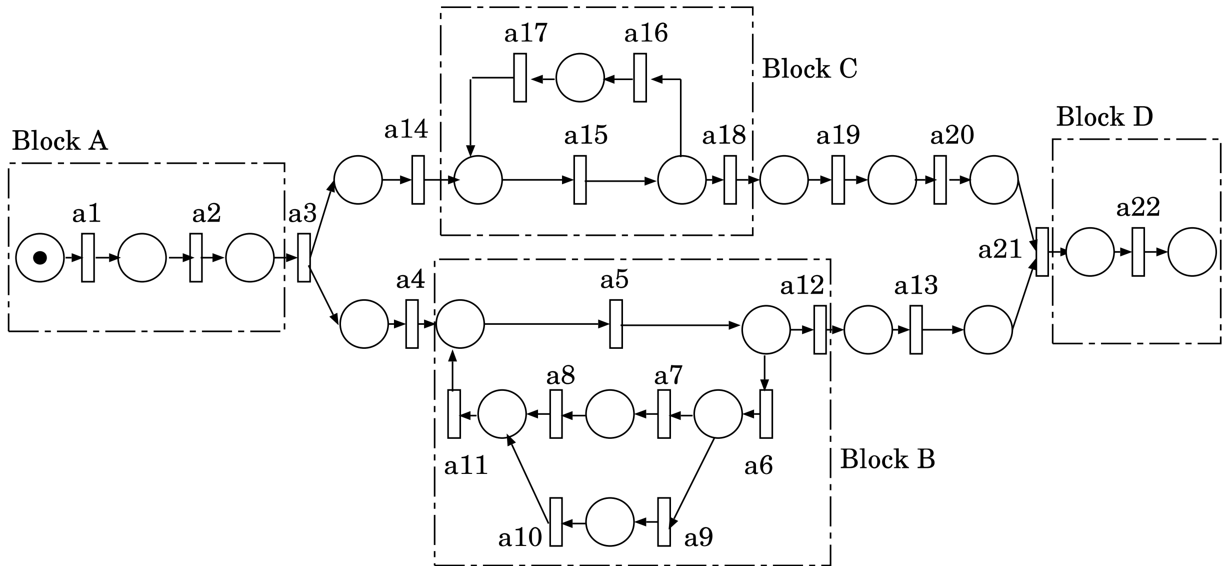


Figure 4.23: Petri net model N_{π_4}.

Σ_1 : If a1 executes, then a13 also executes.
Σ_2 : If a2 executes, then a17 always executes.

Figure 4.24: A specification α for verifying response property.

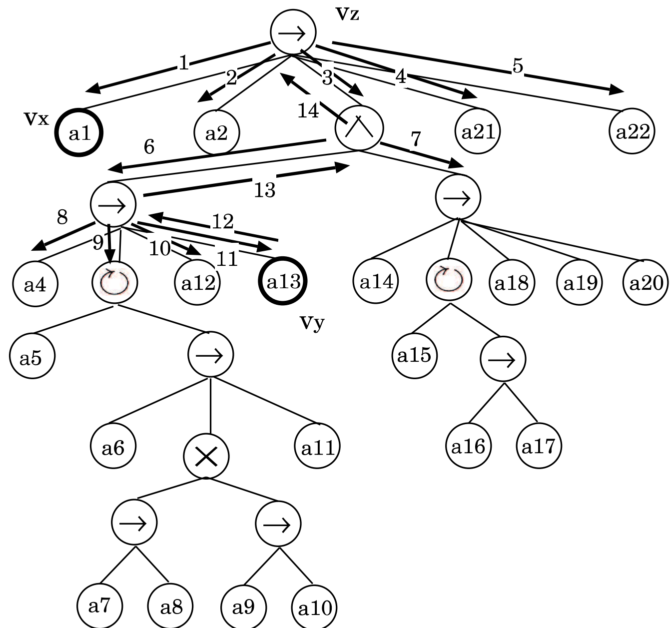


Figure 4.25: Process tree of N_{π_4}. An example to check Σ_1.

4.5 Remarks

In this chapter, we proposed the whole model-driven development for software evolution in details. We first gave the conversion method of Petri net to a representation bias called as process tree. Process tree is a powerful tool to analyze Petri net which can solve state space explotion problem. We showed that the convertibility of WF-net to process tree can be solved in polynomial time. Then, we proposed the convertibility check procedure and discuss its computation complexity. We also proposed a solution to the conversion of WF-net to process tree based on the necessary and sufficient condition where a program structure is a bridge-less well-structure WF-net.

Next, we proposed state number calculation to grasp the scale of a program. It is very important for analysis of software state number analysis. We shows that it is intractable for FC WF-nets and classes bigger than FC. Then we proposed a process tree based state number calculation method for PTB WF-net. The method can be solved in polynomial time. We also proposed a developed tools and evaluated the method. We can calculate more than 9 million states in just a few seconds.

Then, we proposed the method to check behavioral inheritance with Petri net. We can check whether a new program Y is backward compatible with a program Y or not by checking its inheritance. Next, we propose response property analysis method to check execution sequence in a program. Behavioral inheritance check is for minor upgrade and response property check is for major upgrade.

Chapter 5

Other Application and Evaluation

5.1 Security Protocol Implementation and Its Verification

Our model-driven approach for software evolution can be applied to security protocol implementation [29, 42]. As an example, let us take an instance to concretely illustrate the application of a security program that can be embedded to the i-Refrigerator. We take an example of a server program in a cloud environment to gather food data from many i-Refrigerator as the clients. The food data then can be used to automatically order ingredients that are running out or near expiry date.

In this program, security protocol implementation plays an important role. Therefore, we show the application for security protocol implementation in this chapter.

Instance 6
Instance : Security protocol specification α_1 *(See Fig. 5.1), program* π_1 *(See Fig. 5.2), program* $\pi_1{}'$ *(See Fig. 5.3)*
Question : Is α_1 *implemented in program* $\pi_1{}'$ *?* □

Let us consider Instance 6. Is α_1 implemented in program $\pi_1{}'$? π_1 is a program for a server to listen and accept connection from multiple clients. A specification α_1 (See Fig. 5.1) is implemented into program π_1 (See Fig. 5.2). Specification α_1 tells the server program to listens to any client that needs to establish a connection. Then it authenticates the connection. Once the connection is accepted, the protocol requires the server to send an access token to its client so a secure session can be created. Next, the server will handle the connection with the client and closes the connection socket after the session finish. The server can also handle multiple clients.

We illustrate our software evolution as shown in Fig. 1.1. In the *Operation* stage let us say a security consideration was made to a program π_1 after several test operations, the session autho-

a_1 : Create TCP server socket
a_2 : Authenticate connection.
a_3 : Accept TCP connection.
a_4 : Send access token.
a_5 : Handle TCP client.
a_6 : Terminate connection.

Figure 5.1: A security protocol specification α_1.

rization for the clients to start network sessions was not implemented. Moreover, after sessions are finished or failed to initiate, the program could not handle errors or terminate the connection properly. In this case, some connections with the clients that are failed or finished with their sessions are not terminated thus leaving open ports on the server. Hence, in the *Specification* stage we specify an upgrade α_1 as shown in Fig. 5.1. Next, in the *Implementation* stage, we enhanced the program π_1 with the new specification α_1. The new program π_1' is shown in Fig. 5.3. Then, we proceed to the *Verification* stage. It is important to verify whether the specification α_1 is preserved or not in program π_1'.

We apply ≪Backward Compatibility Verification≫ to program π_1 and π_1' given in Fig. 5.2 and Fig. 5.3. In Step 1°, we first translate program π_1 and π_1' into WF-net models N_{π_1} and $N_{\pi_1'}$. The converted models are shown in Fig. 5.5(a)(b). Figure 5.4 shows N_α which is already implemented in N_{π_1}. Next, in Step 2° we check if N_{π_1}' inherits the behavior of N_{π_1}. We apply the backtracking algorithm in Step 2-2° to check if N_{π_1}' is a subclass of N_{π_1} under life-cycle inheritance. The algorithm traverses the reachability tree with depth-first search. Figure 5.6 shows the comparison of reachability tree $R(N_{\pi_1}, [p_I^X])$ and $R(N_{\pi_1}', [p_I^X])$ of N_{π_1} and N_{π_1}'. The dotted lines represent the firing sequences of the markings in the reachability tree which was added in N_{π_1}' after the evolution. $R(N_{\pi_1}', [p_I^X])$ contains all markings and transition sequences which exist in $R(N_{\pi_1}, [p_I^X])$. We obtained that N_{π_1}' inherits the behavior of N_{π_1} because the states in N_{π_1} was preserved in N_{π_1}'. In Step 3°, the procedure outputs yes. We obtained that π_1' implements α. Then, finally the procedure outputs 'yes' and stop.

Another security software development also includes intrusion detection system development [64, 65, 66]. The security software requires accurate implementation of security software. Therefore, besides verifying the security protocol logical correctness, the security protocol implementation is also important.

```
 1 #include <sys/wait.h>
 2 int main(int argc, char *argv[]) {
 3 int servSock, clntSock;
 4 unsigned short echoServPort;
 5 pid_t processID;
 6 unsigned int childProcCount=0;
 7
 8  printf("Server Started...\n");
 9  if(argc != 2){
10    fprintf(stderr,"Port:%s\n",argv[0]);
11     exit(1);
12  }
13  servSock=CreateTCPSocket(atoi(argv[1]);/*a1*/
14  AuthConn(servSock);/*a2*/
15
16   while(1) {
17    clntSock=AcceptTCPConn(servSock);/*a3*/
18    if ((processID=fork()) < 0) {
19       DieWithError ("fork() failed.");
20    }else if (processID=0) {
21       if(servsock>0){
22        close(servSock);
23     HandleTCPClient(clntSock,TOKEN);/*a4,a5*/
24       }
25     exit(0); /*a6*/
26    }
27  }
28 }
```

Figure 5.2: A program π_1 implemented with security protocol α_1.

```
 1 #include <sys/wait.h>
 2 int main(int argc, char *argv[]) {
 3 int servSock, clntSock;
 4 unsigned short echoServPort;
 5 pid_t processID;
 6 unsigned int childProcCount = 0;
 7
 8  printf("Server Started...\n");
 9  if(argc != 3){
10    fprintf(stderr, "Port:%s\n", argv[0]);
11     exit(1);
12   }
13
14   servSock=CreateTCPSocket(atoi(argv[1])); /*a1*/
15
16   if(AuthConn(servSock)==ADMINSOCK){ /*a2*/
17     printf("Admin machine connected...");
18   else if(SESSION_EXPIRED)close(servSock);
19
20   while(1) {
21    clntSock = AcceptTCPConn(servSock); /*a3*/
22    if ((processID= fork()) < 0) {
23       DieWithError ("fork() failed.");
24    }else if (processID=0) {
25       if(servsock > 0){
26        close(servSock);
27        char* token=encrypt(TOKEN); /*a4*/
28      if(!HandleTCPClient(clntSock,token)){ /*a5*/
29         exit(0);
30        }
31       }else{
32         printf("Connection terminated...");
33       }
34     exit(0); /*a6*/
35    }
36     close(clntSock);
37     childProcCount++;
38
39    while(childProcCount){
40      processID = waitpid((pid_t)-1,NULL,WHOANG);
41      if(processID < 0)DieWithError("...");
42      else if(processID == 0) break;
43      else childProcCount--;
44    }
45  }
46 }
```

Figure 5.3: An extended program $\pi_1{}'$ of program π_1.

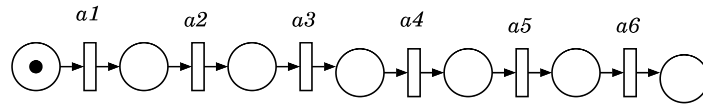
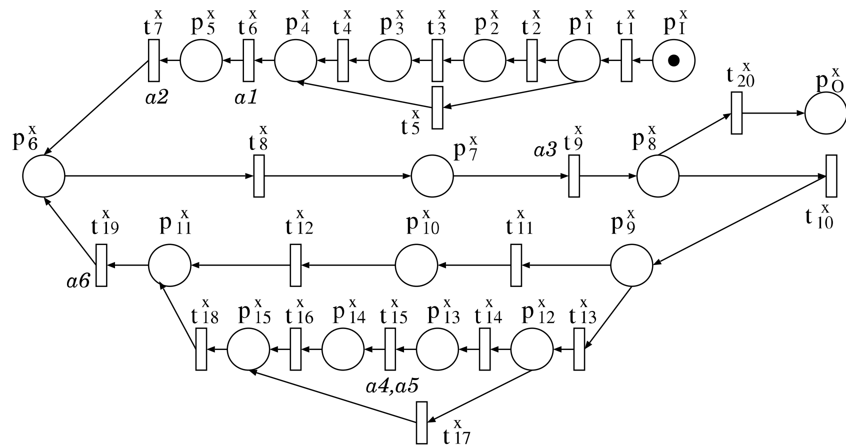
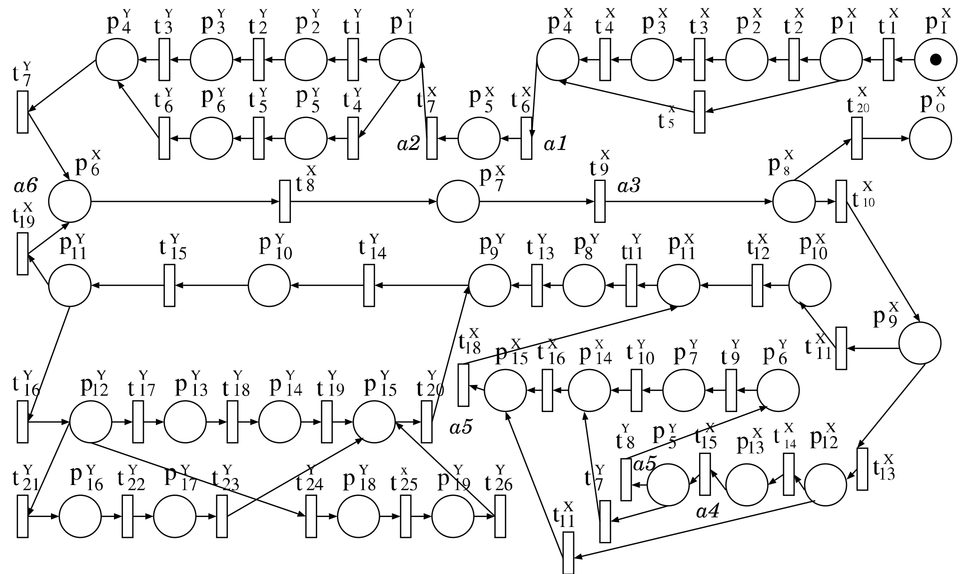


Figure 5.4: Specification α's WF-net N_α.



(a) Program model N_{π_1}.



(b) Implemented model $N_{\pi_1}{}'$.

Figure 5.5: The translated model N_{π_1} and $N_{\pi_1}{}'$. Does $N_{\pi_1}{}'$ inherits the behavior of N_{π_1}?

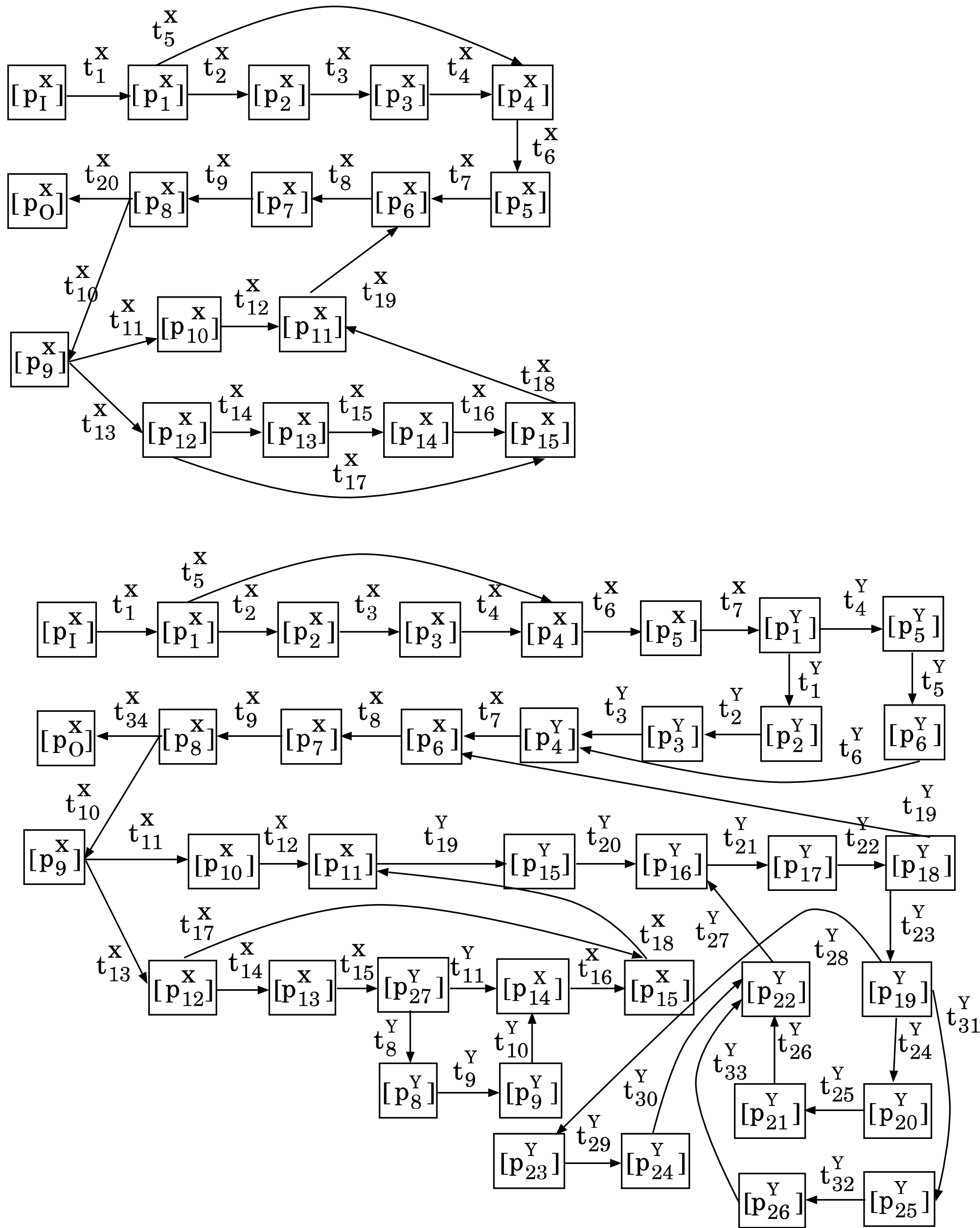


Figure 5.6: Comparison of reachability tree of N_{π_1} (above) and $N_{\pi_1}{}'$ (below) (See Fig. 5.5). $R(N_{\pi_1}{}', [p_I^X])$ contains all markings and transition sequences which exist in $R(N_{\pi_1}, [p_I^X])$. Thus $N_{\pi_1}{}'$ inherits the behavior of N_{π_1}.

5.2 Process Tree Analysis Tool

We evaluate our algorithm with a tool we had developed named Process Tree Analysis Tool (ProTAT) version 2.0 (See Ref. [24]). We can convert a given WF-net to a process tree, then calculate the state number. The tool was developed with Java and Processing 2.0 and is a multi-platform tool.

ProTAT can read Petri Net Markup Language (PTML for short). Fig. 5.7 shows the screen-shot when reading PNML file. Then it can convert PNML file into Process Tree Markup Language (PTML for short). The conversion from PNML to PTML converts Petri net model into process tree as shown in Fig. 5.8. Before the conversion, a convertibility check will verify the class of the Petri net model. If the Petri net model is PTB WF-net, then it converts the model into process tree. Then, we can calculate the state number of the model with ProTAT [63].

We took PTB WF-nets PTB_i (i=$1, 2, \cdots, 20$) as experiment data (See Table 5.1). Figure 5.9 shows PTB_1. PTB_{i+1} was constructed by replacing a place of PTB_i with PTB_1 by refinement [38] to increase the number of parallel paths. For example, place p_7 in PTB_1 can be replaced with PTB_1 itself to produce PTB_2, then PTB_2 can be refinemented with PTB_1 again to produce PTB_3. The evaluation result is shown in Table 1. Based on ProTAT result, the calculation took about 32 seconds for PTB_{20} with over 9 million states.

The experiment was done on Ubuntu Linux with Intel Xeon 2.4 GHz processor and 4 GB memory. Note that calculation time also includes convertibility check time and conversion time.

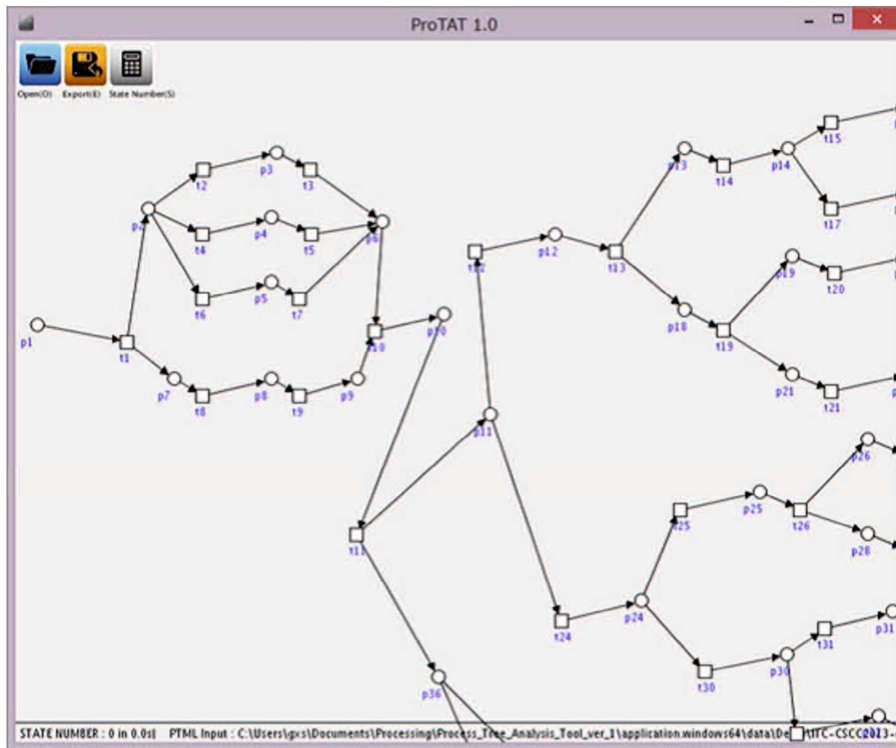


Figure 5.7: Process Tree Analysis Tool version 1.0: Reading Petri net file.

Table 5.1: Size and computation time for PTB WF-net.

WF-net	$\|P\|$	$\|T\|$	$\|P\|+\|T\|$	State Number	Time [s]
PTB_1	8	7	15	10	0.021
PTB_2	15	14	29	28	0.041
PTB_6	43	42	85	568	0.246
PTB_8	57	56	113	2,296	0.526
PTB_{12}	85	84	169	36,856	3.333
PTB_{14}	99	98	197	147,448	6.571
PTB_{18}	127	126	253	2,359,288	21.205
PTB_{20}	141	140	281	9,437,176	32.319

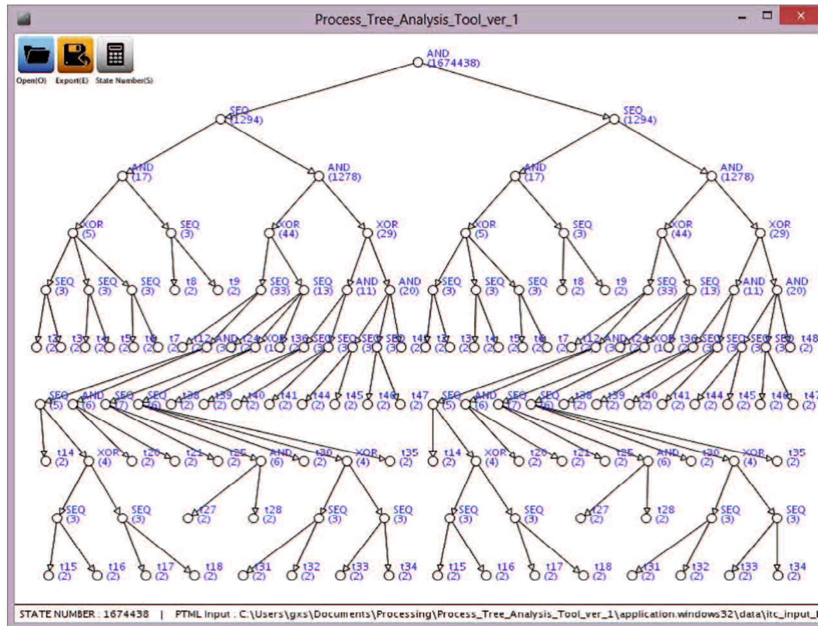


Figure 5.8: Process Tree Analysis Tool version 1.0: Conversion to process tree.

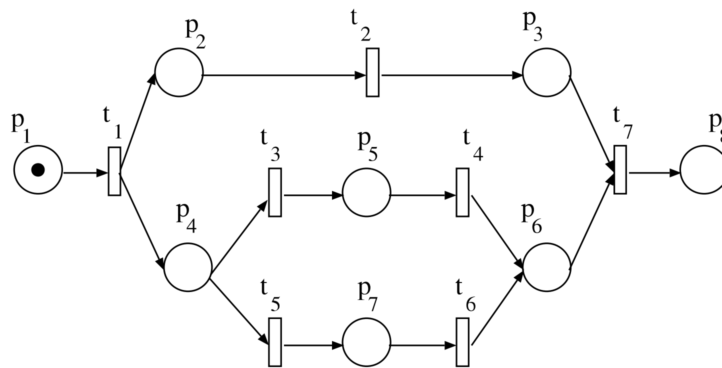


Figure 5.9: PTB WF-net PTB_1.

5.3 Remarks

In this section, we showed another application for our model-driven development method. We whosed that security protocol implementation throughout the evolution with our backward compatibility check.

We also proposed a tool called as Process Tree Analysis Tool that can convert a Petri net into process tree and calculate the state number. We showed the result to calculate more than 9 million states where state space explosion can occur. The tool can run in practical time.

Chapter 6

Conclusion

In software evolution, software changes through repetitive development stages. This cause validation and management of product family line to be difficult. Therefore, we proposed a model-driven development method for software evolution that preserve the product family line.

Petri nets have been widely used for analysis of software. Once we model a system as a Petri net, we can simulate the behavior of the system by using tokens on the Petri net. We can also analyze the behavior of the system exhaustively by enumerating all possible token distributions (states). Unfortunately, the number of all the possible states is of exponential order in the size of the Petri net. We called the problem as state space explosion. In conventional ways, if we enumerate all possible markings, it is intractable to count the number of nodes in the reachability tree. In software development, we need a method to analyze any properties in a program such as state number and response property. So we proposed the utilization of process tree.

In Chap. 2 we introduced software evolution, backward compatibility and the definition of Petri net, workflow net, ST-net and process tree. We also gave an example and diagram of workflow net and process tree. We introduced our study for software evolution which focus on the backward compatibility of software in the evolution. We also introduced the backward compatibility and its definition. Backward compatibility is also related to behavioral inheritance where we gave the formal definition of behavioral definition such as life-cycle inheritance in the preliminary.

Then, in Chap. 3 we discuss real world problem of software evolution. We know that during the evolution of software, it is important to manage and preserve the product line of the software. As software evolves to adapt to requirements of consumer over time, the newer version of the software tends to become more complex. We showed that we can analyzed software backward compatibility with Petri net. The, we proposed a reverse engineering method to translate program into Petri net model. We also showed that a class of Petri net can represent program structure

called as bridge-less WF-net. We revealed the property for the Petri net class.

Next, in Chap. 4 we proposed the whole model-driven development for software evolution in details. We first gave the conversion method of Petri net to a representation bias called as process tree. Process tree is a powerful tool to analyze Petri net which can solve state space explosion problem. We showed that the convertibility of WF-net to process tree can be solved in polynomial time. Then, we proposed the convertibility check procedure and discuss its computation complexity. We also proposed a solution to the conversion of WF-net to process tree based on the necessary and sufficient condition where a program structure is a bridge-less well-structure WF-net. This shows that our converted program is represented as sound WF-net where there are many analysis techiques available for program analysis.

Next, we proposed state number calculation to grasp the scale of program. It is very important for analysis of software state number analysis. We shows that it is intractable for FC WF-nets and classes bigger than FC. Then we proposed a process tree based state number calculation method for PTB WF-net. The method can be solved in polynomial time. We also proposed a developed tools and evaluated the method. We can calculate more than 9 million states in just a few seconds. We showed that we can avoid state space explosion with out method.

Then, we proposed the method to check backward compatibility with Petri net. We utilize a model-based algorithm to check whether a new program Y inherits the behavior of program Y or not. Next, we propose response property analysis method to check execution sequence in a program. Behavioral inheritance check is for minor upgrade and response property check is for major upgrade.

In Chap. 5 we proposed another application of process tree in security protocol implementation. Security protocol implementation is very important for the development of secure program particularly related to embedded program that shares data on the network. We showed the example for the security program of the i-refrigerator. Then, we showed our developed tool called as Process Tree Analysis Tool and evaluated the tool.

As the future work, we would like to apply the method to various software development including web programs, embedded programs and security software development. Also, we would like to extend the research work to another property such as forward compatibility.

Bibliography

[1] C. Hibbs, S. Jewett, M. Sullivan, *The Art of Lean Software Development: A Practical and Incremental Approach*, O'Reilly Media, Inc, 2009.

[2] M.A.B. Ahmadon, S. Yamaguchi, "A Petri net based support for derivative development of consumer electronic products," Proc. of IEEE GCCE 2013, pp.212–216, 2013.

[3] M.A.B. Ahmadon, S. Yamaguchi, "Tailor made device driver design system based on Petri net," Proc. of IEEE GCCE 2014, pp.703–706, 2014.

[4] I. Sommerville, *Software Engineering* (9th ed.), Addison-Wesley, 2011.

[5] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1998.

[6] M. Brambilla, J. Cabot, M. Wimmer, *Model-Driven Software Engineering in Practice* (2nd ed.), Morgan & Claypool Publishers, 2012.

[7] J. Sang, D. Hong, B.Zhang, L. Fu "Protection profile for the smartphone operating system," Int. J. of Embedded Systems, vol.6, no.1, pp.28–37, 2014.

[8] L. Tawalbeh, Y.Haddad, O.Khamis, E.Benkhelifa, Y.Jararweh, F.AlDosari, "Efficient and secure software-defined mobile cloud computing infrastructure," Int. J. of High Performance Computing and Networking, vol.9, no.4, pp.328–341, 2016

[9] B. Rudra,A.P. Manu, O.P. Vyas, "Service-oriented network architecture: significant issues and principles of communication," Int. J. of Computational Science and Engineering, vol.10, no.3, pp.306–314, 2015

[10] M. Avalle, A. Pironti, R.Sisto, "Formal verification of security protocol implementations: a survey," Formal Aspects Computing, vol. 26, no. 1, pp. 99–123, 2014.

[11] Y. Moffett, J. Dingel, A. Beaulieu, ”Verifying Protocol Conformance Using Software Model Checking for the Model-Driven Development of Embedded Systems,” IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1307–1325, 2013.

[12] Y.Fu, O.Kon,“Model based security verification of protocol implementation,” Journal of Information Security and Applications,Vol.22, pp.17–27,2015

[13] S.Smith, A.Beaulieu, W.G.Phillips,“Modeling and verifying security protocols using UML 2,” Systems Conference(SysCon), IEEE International, Montreal, QC, pp.72–79,2011

[14] J.Goubault-Larrecq, F.Parrennes,“Cryptographic protocol analysis on real C code,” Proc. of the 6th international conference on verification, model checking, and abstract interpretation (VMCAI),pp. 363–379,2005

[15] S.Chaki, A.Datta, “ASPIER: an automated framework for verifying security protocol implementations,” Proc. of the 22nd IEEE symposium on computer security foundations (CSF), pp. 172–185,2009

[16] T.Murata,B.Shenker, S.M.Shatz,“Detection of Ada static deadlocks using Petri net invariants,” IEEE Trans. on Software Engineering, vol.15, no.3, pp.314–326,1989

[17] O.F.Rana, M.S.Shields, S.M.Shatz, “Performance Analysis of Java Using Petri Net,” Proc. HPCN Europe 2000, pp.657–667,2000

[18] J.Voron, F.Kordon, “Transforming sources to petri nets: a way to analyze execution of parallel programs,” Proc. of SimuTools 2008, vol.13, 2008

[19] J.Voron, F.Kordon : Evinrude: A Tool to Automatically Transform Programs Sources into Petri Nets, *Petri Net Newsletter*, vol. 75, pp. 19–38, 2008

[20] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004

[21] G. Regis, F. Villar, N. Ricci, “Fluent Logic Workflow Analyser: A Tool for The Verification of Workflow Properties,” Proc. of First Latin American Workshop on Formal Methods (LAFM), Buenos Aires, pp.46–51, 2013

[22] M.S.B.A. Malek, M.A.B. Ahmadon, S. Yamaguchi, B.B. Gupta, “Implementation of parallel model checking for computer-based test security design,” Proc. of 7th International Conference on Information and Communication Systems (ICICS), Irbid, pp. 258–263, 2016

[23] M.A.B.Ahmadon, S.Yamaguchi, B.B.Gupta, “A Petri-net based approach for software evolution,” Proc. of the 7th Int. Conf. on Information and Communication Systems (ICICS), Irbid, Jordan, pp.264–269, 2016

[24] M. A. Bin Ahmadon, S. Yamaguchi, “Convertibility and Conversion Algorithm of Well-Structured Workflow Net to Process Tree,” Proc. CANDAR 2013, pp.122–127, 2013.

[25] M.A. Bin Ahmadon, S. Yamaguchi, “State Number Calculation Problem of Workflow Nets,” IEICE Trans. Infomation and System, vol.E98–D, no.6, pp.1128–1136, 2015.

[26] S. Erdweg, S. Fehrenbach, K. Ostermann, ”Evolution of Software Systems with Extensible Languages and DSLs,” IEEE Software, vol. 31, no. 5, pp. 68–75, 2014.

[27] T. D. Oyetoyan, D. S. Cruzes, R. Conradi, ”Transition and defect patterns of components in dependency cycles during software evolution,” 2014 Software Evolution Week - IEEE CSMR-WCRE, pp. 283–292, 2014.

[28] I. Toyoshima, S. Yamaguchi, Y. Murakami, ”Two sufficient conditions on refactorizability of acyclic extended free choice workflow nets to acyclic well-structured workflow nets and their application,” IEICE Trans. Fundamentals, vol.E98-A, no.2, pp.635–644, 2015.

[29] M.A.B. Ahmadon, S. Yamaguchi, B.B. Gupta, “Petri Net-Based Verification of Security Protocol Implementation in Software Evolution,” Int. J. of Embedded Systems, 2016. (in-print)

[30] T. Mens, ‘Introduction and roadmap: History and challenges of software evolution,’ *Software Evolution*, Springer-Verlag, pp. 1–11, 2008.

[31] C.A. Petri, *Communication by automata*, PhD of the University of Technology Darmstadt, Germany, 1962.

[32] T. Murata, “Petri nets: Properties, analysis and applications,” Proceedings of the IEEE, vol.77, no.4, pp.541–580, 1989.

[33] S. Yamaguchi, M.A.B. Ahmadon, Q.W. Ge, “Introduction of Petri Nets: Its Applications and Security Challenges,” *Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security*, Hershey, PA, USA, chapter 7, pp.145–179, 2016.

[34] W.M.P. van der Aalst, “Verification of workflow nets,” LNCS, vol.1248, pp.407–426, 1997.

[35] J. Esparza, M. Silva, “Circuits, handles, bridges and nets,” Lecture Notes in Computer Science, vol.483, pp.210–242, 1990.

[36] J. Desel, J. Esparza, *Free Choice Petri Nets*, Cambridge University Press, 1995.

[37] W.M.P. van der Aalst, K. van Hee, *Workflow Management: Models, Methods, and Systems*, The MIT Press, 2002.

[38] K. M. van Hee, N. Sidorova, M. Voorhoeve, “Soundness and Separability of Workflow Nets in the Stepwise Refinement Approach,” Proc. ICATPN 2003, vol.2679, pp.337–356, Springer-Verlag, Berlin, 2003.

[39] W.M.P. van der Aalst, T. Basten, “Inheritance of workflows: An approach to tackling problems related to change,” Theoretical Computer Science, vol.270, no.1–2, pp.125–203, 2002.

[40] H.M.W. Verbeek, T. Basten, “Deciding life-cycle inheritance on Petri nets,” Proc. of ICATPN 2003, pp.44–63, 2003.

[41] M.A.B. Ahmadon, S. Yamaguchi, B.B. Gupta, “A Petri-Net Based Approach for Software Evolution,” Proc. of IEEE ICICS 2016, pp.264–269, 2016.

[42] W. Tang, Z. Gou, M.A.B. Ahmadon, S. Yamaguchi, “On Verification of Implementation of Security Specification with Petri Nets’ Protocol Inheritance,” Proc. of IEEE GCCE 2016, pp.497–500, 2016.

[43] A. Konno, Y. Masunaga, “Concept of an Intelligent Refrigerator System using RFID,” The database society of Japan letters, vol.4, no.2, pp.73–76, 2005.

[44] Code2Flow, https://code2flow.com/, 2016.

[45] Code Visual to Flowchart, http://code-visual-to-flowchart-full-version.software.informer.com/, 2011.

[46] AutoFlowchart, http://autoflowchart.software.informer.com/3.5/, 2011.

[47] J.R.Levine, T.Mason, D.Brown, *Lex and Yacc*, O’Reilly and Associate, 1998.

[48] J.Lee, ANSI C Yacc grammar, http://www.lysator.liu.se/c/ANSI-C-grammar-y.html, 1985.

[49] H. Taniguchi, S. Yamaguchi, T. Susaki, “A Translator of C to Petri Net Markup Language C2PNML and Its Application Examples,” IEICE Tech. Rep., vol.111, no.453, MSS2011-78, pp.35–40, 2013.

[50] S. Yamaguchi, M.A.B. Ahmadon, ”Properties and decision procedure for bridge-less workflow nets,” IEICE Trans. Fundamentals, vol.E99-A, no.2, pp.509-512, 2016.2.

[51] S. Yamaguchi, “Polynomial time verification of reachability in sound extended free-choice workflow nets,” IEICE Trans. Fundamentals, vol.E97-A, no.2, pp.468–475, 2014.

[52] J. Esparza, M. Nielsen, “Decidability issues for Petri nets,” Bulletin of the EATCS 52, pp.244–262, 1994.

[53] J. Esparza, M. Silva, “Circuits, handles, bridges and nets,” Lecture Notes in Computer Science, vol.483, pp.210–242, 1990.

[54] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, The MIT Press, pp.998–1003, 2001.

[55] R. E. Tarjan, “Depth-first Search and Linear Graph Algorithms,” SIAM J. Comput.1, no.2, pp.146–160, 1972.

[56] D.Y. Chao, Y. Fang, ”Number of Reachable States for Simple Classes of Petri Nets,” Proc. of IECON 2011, pp.3788–3791, 2011.

[57] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, Second Edition, The MIT Press, pp.998–1003, 2001.

[58] R.R. Howell, L.E. Rosier, H. Yen, “Normal and sinkless Petri nets,” J. Computer and System Sciences, vol.46, pp.1–26, 1993.

[59] A. Ohta, K. Tsuji, “NP-hardness of liveness problem of bounded asymmetric choice net,” IEICE Trans. Fundamentals, vol.E85-A, no.5, pp.1071–1075, 2002.

[60] S. Yamaguchi, M. Yamaguchi, M. Tanaka, “A model checking method of soundness for acyclic workflow nets using the SPIN model checker,” Int. Journal INFORMATION, vol.12, no.1, pp.163–172, 2009.1.

[61] O.E. Hichami, M.A. Achhab, I. Berrada, R. Oucheikh, B.E.E. Mohajir, “An Approach of Optimization and Formal Verification of Workflow Petri Nets,” J. Theoretical and Applied Information Technology, vol.61, no.3, pp.486–495, 2014.

[62] S. Yamaguchi, T. Hirakawa “Polynomial time verification of protocol inheritance between acyclic extended free-choice workflow nets and their subnets,” IEICE Trans. Fundamentals, vol.E96-A, no.2, pp.505–513, 2013.

[63] M.A.B. Ahmadon, S. Yamaguchi, "On state number calculation in Petri nets," Proc. CANDAR 2014, pp.116-122, 2014.12.

[64] Z. Gou, M.A.B. Ahmadon, S. Yamaguchi, B. Gupta, "A Petri Net-based Framework of Intrusion Detection Systems," Proc. of IEEE GCCE 2015, pp.579–583, 2015.

[65] M.A.B. Ahmadon, Z. Gou, Shingo Yamaguchi, B.B. Gupta, "Detection and Update Method for Attack Behavior Models in Intrusion Detection Systems," Proc. of INDIACom 2016, pp. 5637–5642, 2016.

[66] M.S.B.A. Malek, M.A.B. Ahmadon, S. Yamaguchi, B. Gupta, "On Privacy Verification in the IoT Service Based on PN2," Proc. of IEEE GCCE 2016, pp.8–11, 2016.