FINAL REPORT
FUNDAMENTAL RESEARCH GRANT SCHEME (FRGS)
Laporan Akhir Skim Geran Penyelidikan Fundamental (FRGS)

SEQUENCE DIAGRAM SYNTHESIS THEORY FOR STITCHING TOGETHER MULTIPLE SEQUENCE DIAGRAMS THROUGH FORMAL SEMANTICS MODELLING

by

PROJECT LEADER:
MOHAMED ARIFF BIN AMEEDEEN

PROJECT MEMBERS:
KAMAL ZUHAIRI BIN ZAMLI
MUHAMAD IDAHAM BIN UMAR ONG
THONG WENG JIE

CHAPTER 1

INTRODUCTION AND GENERAL INFORMATION

This chapter serves as an introduction to the thesis. Throughout this chapter, subtopics such as purpose of the thesis, background of study, problem statement, research objective, research scope, research outcome and organization of the thesis will be presented.

1.1 PURPOSE OF THE THESIS

The aim for this thesis is to bridge the gap between Sequence Diagram, Petri Nets and SD2PN. Sequence Diagram is a behavioral type of UML diagram widely used by software developers to show dynamic interactions in a system, while Petri Net is a modelling language that is able to carry out mathematical analysis for a system that is also capable of expressing dynamic interaction in a system. SD2PN is a tool that enables software developers to map Sequence Diagram to Petri Nets. Software developers are able to map Sequence Diagram to Petri Nets and perform mathematical analysis using Petri Net tools to check for any error before the actual coding phase. This will in turn result in less error during the coding phase. However, SD2PN only supports one way mapping, which is from Sequence Diagram to Petri Nets. Users have to manually update the Sequence Diagram if any error is to be found when analyzing the Petri Nets. This thesis aims to find a way to map Petri Nets back to Sequence Diagram, so that software developers are able to map the Petri Nets back to Sequence Diagram instead of updating the Sequence Diagram manually. This is the main motivation for this thesis, which is to create an algorithm for mapping Petri Nets to UML Sequence Diagram. This thesis is also been done to fulfill the requirement of my masters study.

1.2 BACKGROUND OF STUDY

Software engineering is the application of engineering technique to design, develop and maintenance for the software. Software engineering can be divided into ten sub disciplines which are software requirements, software design, software construction, software testing, software maintenance, software configuration management, software engineering management, software engineering process, software engineering tools and methods and software quality [1]. Software design is one of the vital stages of a software development life cycle. It is the blueprint of the design and architecture of the software. Implementation of the coding will refer to the software design. Hence, it is important for the software design to have as minimal error as possible as it might lead to errors during implementation phase.

In software design, the designer creates specifications of a software artifact based on the requirements given by stakeholders [2]. It usually involves problem solving and planning a software solution. It is important to have as minimal error as possible in the software design phase. Hence, languages in the form of models like Petri Nets [3] are used because of its ability to perform formal, mathematical analysis of the software designs. These formal modelling languages are able to perform mathematical analysis of the software design such as liveness, deadlock detection and reachability. This in turn will be able to reduce the amount of design errors being carried into the implementation phase.

Petri Nets is a formal modeling language that can be represented graphically with a strong mathematical foundation [4]. It is represented graphically in the sense that it serves as a visual communication aid to model the system behavior. It is based on a mathematical foundation in the sense that it represents the equations, algebraic equations and algorithms in the systems. Petri Nets can be used to model control flow in a system and is capable of modeling diverse set of parallel, asynchronous, concurrent, hierarchical, stochastic as well as dynamic behaviors.

However, the Unified Modelling Language (UML) is the go-to modelling language for software designers. UML is a general informal modelling language used to describe the software both structurally and behaviorally [5]. UML diagrams can be classified into structural and behavioral diagrams. An example of the structural diagram is the Class Diagram, it models the classes in the systems, attributes and operations and how are they related to each other. While an example of behavioral diagram is the Sequence Diagram which models the dynamic interactions in terms of messages passed between objects in the systems.

SD2PN is a tool for transforming Sequence Diagram into Petri Nets. There are similar properties between Sequence Diagram and Petri Nets where both is able to model dynamic interactions and behaviors in terms of messages passed between objects in the system [6]. SD2PN provides a tool for transforming Sequence Diagram to Petri Nets. This is useful for software designers to transform the Sequence Diagram into Petri Nets and performs mathematical analysis using Petri Nets tools such as CoopnBuilder [7], GreatSPN [8] and Petruchio [9]. However, there are some limitations to the SD2PN tool which will be discussed in the following part.

1.3 PROBLEM STATEMENT

There appears to be a gap between the knowledge of Sequence Diagram, Petri Nets and SD2PN. SD2PN is only capable of performing one way mapping which is from Sequence Diagram to Petri Nets. This will lead to tedious repeated modeling each time a change has been made. Users need to manually update the Sequence Diagram each time a change has been made to the Petri Nets model.

SD2PN partially solved the problem of heterogeneity between Sequence Diagram and Petri Nets. Software developers can use SD2PN to perform model transformation from Sequence Diagrams to Petri Nets. Since it is only a one-way mapping process, the communication is only from Sequence Diagram to Petri Nets. This also means that Petri Nets are not able to communicate with Sequence Diagram. Hence, users are still required to update the Sequence Diagram manually after performing analysis on the Petri Nets generated. It is not a fully automated or a reversible process.

This study will address the problem of SD2PN, which is it only supports one way transformation from Sequence Diagram to Petri Nets. A solution will be look into to create an algorithm for mapping Petri Nets to Sequence Diagram. A standalone tool or an algorithm might be the outcome of the study to map Petri Nets models to Sequence Diagram models.

1.4 RESEARCH OBJECTIVES

- To create an algorithm to transform Petri Nets to Sequence Diagram
- To develop a rule-based MDD model transformation that transforms Petri Nets fragments to Sequence Diagram fragments.
- To show correctness of the model transformation based on existing frameworks.

The research objectives were made based on the problem statement state in Section 1.3. As stated in the problem statement, SD2PN has a limitation of one way transformation, which is from Sequence Diagram to Petri Nets. This is tackled by the first objective, which is to create an algorithm that transforms Petri Nets to Sequence Diagram. While transforming a model to another model, a rule-based Model Driven Development (MDD) model transformation needs to be created, which leads to the second objective of this research. Upon successfully transforming Petri Nets into Sequence Diagram, a common semantic domain is needed to compare if the semantics of the models are preserved. This is shown in the third objective which is to show the correctness of the model transformation based on existing frameworks.

1.5 RESEARCH SCOPES

In order to achieve the outlined objectives, the specific scopes of this research are:

1. To study the techniques used in SD2PN.
2. To study Petri Nets and UML Sequence Diagram.
3. To introduce a way to identify fragments in Petri Nets.
4. To introduce a rule-based Model Driven Development model transformation to represents Petri Nets fragments as Sequence Diagram fragments.
5. To develop an algorithm to transform Free Choice Petri Nets into UML Sequence Diagram.
6. To proof that PN2SD preserves semantics.

1.6 RESEARCH OUTCOMES

As a summary, the main aims of the research are as follows:

The research focuses on improving the limitation of SD2PN and aims to introduce an algorithm to transform Petri Nets to Sequence Diagram. A new method to identify fragments in Petri Nets is introduced and a rule-based MDD model transformation to represent Petri Net fragments as Sequence Diagram is utilized. Other than that, relevant publication will be produced. This thesis explains in detail how the algorithm works, how the fragments in Petri Nets are identified and how is Petri Net fragments represented as Sequence Diagram fragments.

1.7 ORGANIZATION OF THESIS

The remainder of the thesis will be structured as follows:

Chapter 1: This chapter focuses mainly on the brief introduction to software design, Petri Nets, Sequence Diagram and SD2PN. Besides that, this chapter also introduces the limitation of SD2PN.

Chapter 2: In this chapter, preliminary and some basic foundation information on Petri Nets, Petri Nets tools, Sequence Diagram and SD2PN will be presented. Similar works such as UML2Alloy and UML-B / U2B are also introduced.

Chapter 3: The research methodology on the concept of Multi Paradigm Modelling is presented in this chapter. Other than that, the concept of Labelled Event Structures is also introduced in this chapter.

Chapter 4: PN2SD is presented in this chapter. A new method is introduced in identifying fragments in Petri Nets, and the transformation rules to transform Petri Nets fragments into Sequence Diagram fragments is presented in this chapter. The full algorithm to transform Free Choice Petri Nets to Sequence Diagram is also presented in this chapter. This chapter also shows how PN2SD preserves semantics.

Chapter 5: This chapter shows how the algorithm can be applied on a Free Choice Petri Net. An example is given which involves the behavior of a Personal Area Network (PAN). A discussion is also available in this chapter.

Chapter 6: This chapter presents the conclusion, contribution and limitation of the research work. Recommendations for future work are also available in this chapter.

CHAPTER 2

FOUNDATION

This chapter presents preliminary information also known as the basic foundation of the language and the technology used throughout this thesis which includes Petri Net, Petri Net tools, UML, Sequence diagram, UML Tools, SD2PN and also some similar works.

2.1 PETRI NETS

Petri Net is a formal modeling language that can be represented graphically with a strong mathematical foundation [10]. Petri Nets are mostly used to model control flow in a system and is capable of modeling conflicts and concurrencies. There are four main components of a Petri Net, which are places, transitions, arcs and tokens.

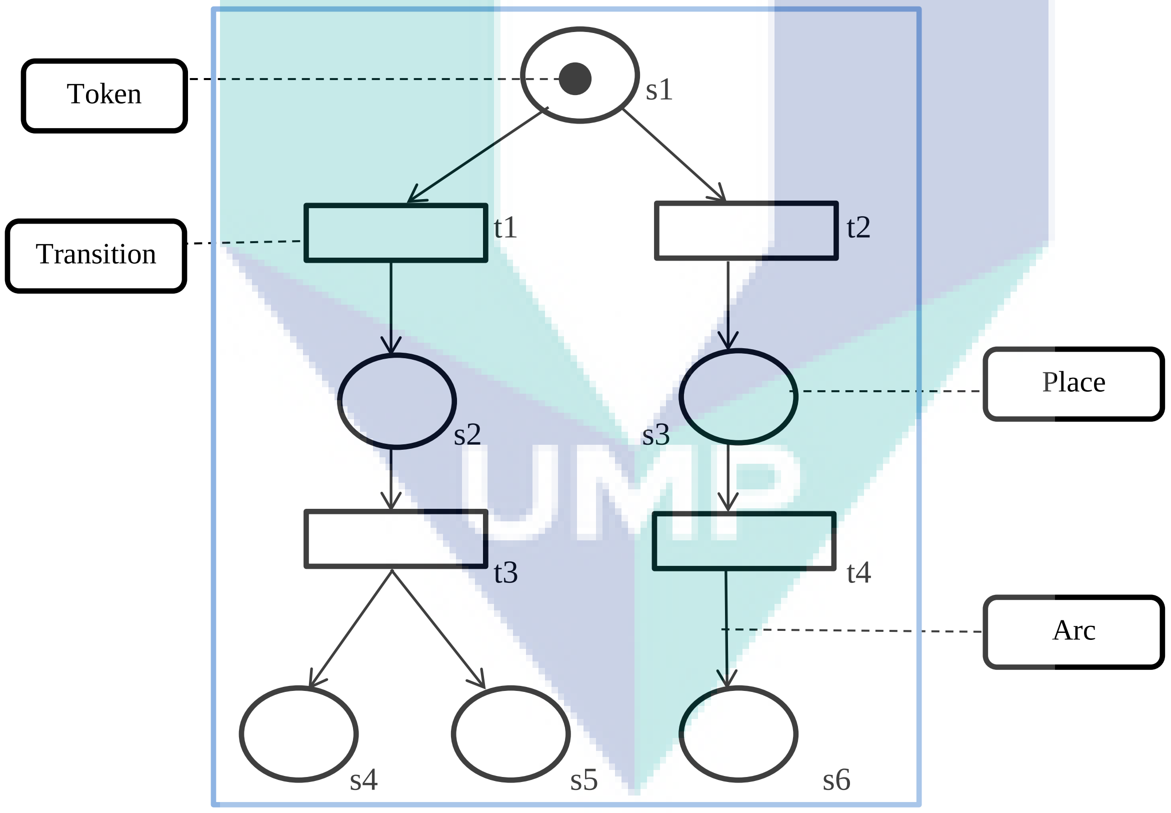


Figure 1: Example of Petri Net

Figure 1 is an example of a Petri Nets with six places, four transitions and nine connecting arcs. A place in Petri Net may contain any number of tokens. A place must be connected to a transition via an arc or vice versa, while an arc can be classified as either an input arc or an output arc. An input arc is characterized as an arrow with the arrowhead pointing towards the place, while an output arc is characterized as an arrow with the arrowhead pointing away from the place. The precondition for a transition to be enabled or ready to fire is that the place connected to the transition needs to be marked with at least one token. In figure 1 case, transition t1 and t2 is enabled and ready to fire, while transition t3 and t4 do not have a marked place with token connected to them, hence t3 and t4 is not enabled and not ready to fire. The firing of the token will be further illustrated in figure 2.

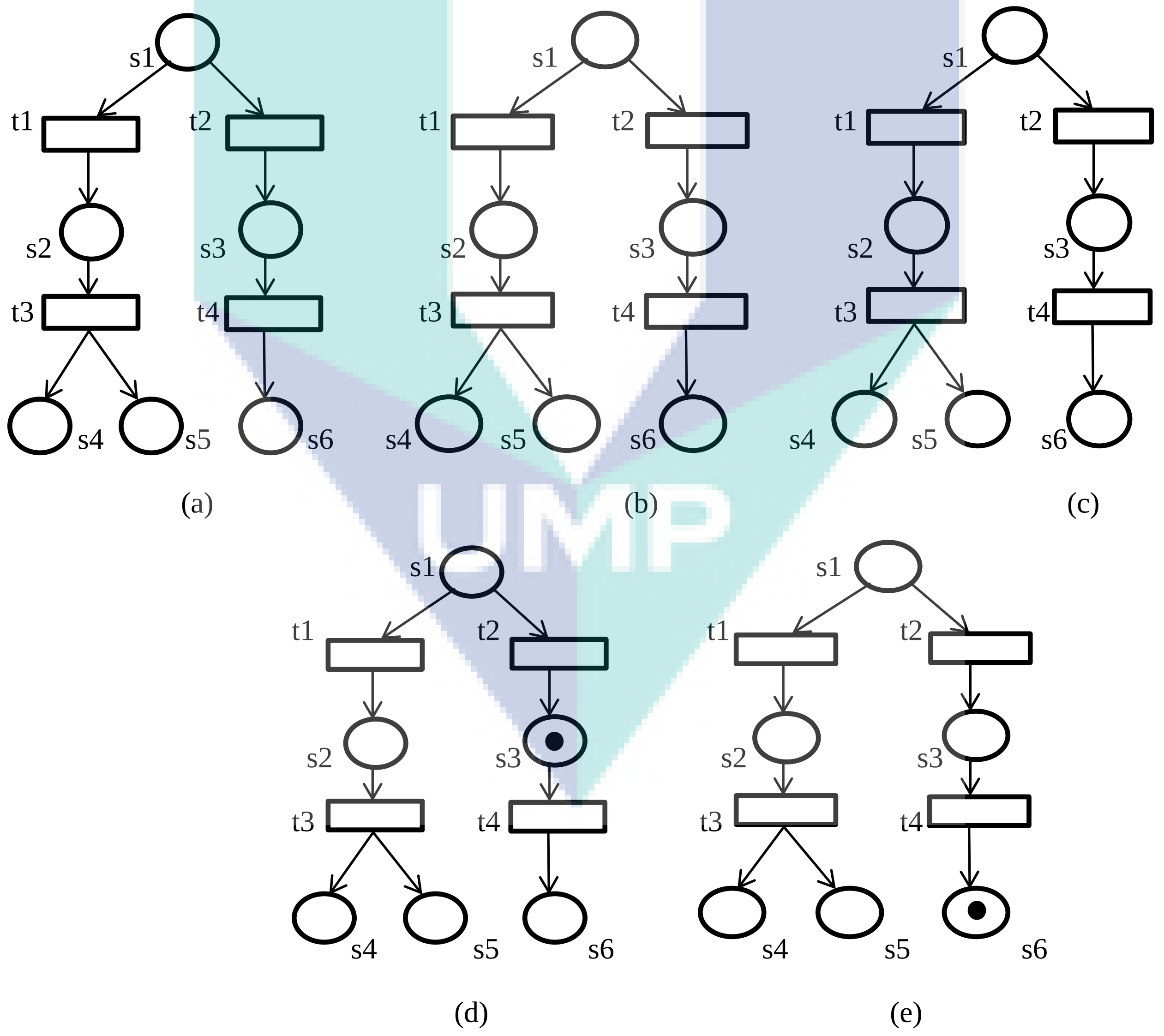


Figure 2: Example of Petri Net token firing sequence

Figure 2 (a) shows a marked place, s1 with output arcs connecting to transition t1 and t2. Hence both t1 and t2 becomes enabled since the place that is connected to them via input arcs are marked. Both t1 and t2 are now enabled but only one of them may fire, this shows a conflict. Figure 2 (b) shows the scenario where t1 fires the token to s2. This action removes the token from s1 and places the token in s2 which is connected via an output arc to t1. The transition t2 is now no longer enabled as s1 is not the marked place anymore. As a result, s2 is now a marked place and t3 is now enabled and ready to fire. In Figure 2 (c), since t3 is the only transition connected to s2, t3 fires the token from s2 and places a token each in s4 and s5. This is defined as a concurrency or parallel relationship where one token is split into two or more (depending on the number of concurrent nodes). Figure 2 (d) shows an alternative scenario where t2 fires the token instead of t1. The token in s1 is removed and placed in s3. The transition t4 now becomes enabled and the firing sequence is continued as there are no conflicts. In Figure 2 (e), the transition t4 fires the token from s3 into s6. The firing of t4 will only occur following the firing of t2. This creates a causal relationship between t2 and t4.

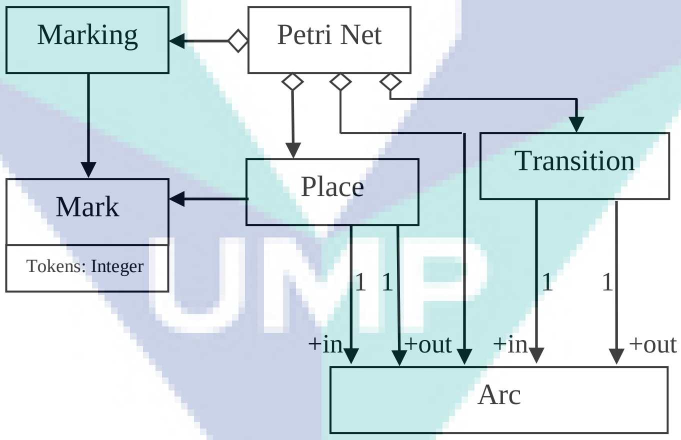


Figure 3: Petri Net Metamodel

Figure 3 shows the metamodel of Petri Net. Each Petri Net consists of at least one place, one transition, one arc and one marking. Each place or transition can have any number of input and output arcs and each place has a mark in the form of the integer number of tokens. Petri Nets can also be represented formally as below:

A Petri Net is a triple N = (S, T, F), where S is a finite set of places and T is a set of transition where $S \cap T = \emptyset$. F is a relation on $S \cup T$ where $F \cap (S \times S) = F \cap (T \times T) = \emptyset$. A marking of N is a function $\mathbf{m}:S \rightarrow \{0,1,2,3, \ldots\}$, where each place $s \in S$ is assigned the number of tokens. M_0 is used to show the initial marking, the number of tokens in each place at the beginning of execution.

2.1.1 PETRI NETS TOOLS

Petri Nets tools are software tools used to design and perform analysis on Petri Nets models. With Petri Nets tools, users are able to analyze the performance of the system. Users are also able to use Petri Nets tools as a graphical editor or code generator. Some of the Petri Nets tools can also be used to simulate the system and provide model checking for it. Based on a recent survey [11], the table below shows a comparison between twenty Petri Nets tools in terms of Petri Nets supported, components, type of analysis, environment to be ran in and price.

Petri Net Tool	Petri Net Supported								Component												Analysis					Environments									
	High-level Petri Nets	Object-oriented Petri Nets	Stochastic Petri Nets	Petri Nets with Time	Place/Transition Nets	Continuos Petri Nets	Transfer Petri Nets	Queueing Petri Nets	Graphical Editor	State Spaces	Condensed State Spaces	Code Generatin	Token Game Animation	Fast Simulation	Place Invariants	Transition Invariants	Net Reduction	Model Checking	Petri Net Generator	Interchange File Format	Simple Performance Analysis	Structural Analysis	Advance Performance Analysis	Reachability Graph Based Analysis	Invariant Based Analysis	Java	Linux	Sun	HP, HP-UX	Silicon Graphics, IRIX	MS DOS	Windows	Macintosh	UNIX	Free of Charge
AlPiNA	x								x	x	x									x	x					x									x
CoopnBuilder	x	x							x			x	x	x												x									x
GreatSPN	x		x	x					x	x	x		x	x	x	x						x	x				x	x							x
LoLA	x				x					x	x			x													x	x	x	x	x	x			x
PEP	x			x	x				x	x	x		x		x	x	x	x	x	x		x					x	x							x
SNOOPY			x	x	x	x			x				x	x													x					x	x		x
MARCIE			x							x								x									x						x		x
CHARLIE			x		x	x			x									x				x		x	x	x	x					x			x
JSARP		x							x				x	x								x				x									x
MIST					x					x	x				x	x	x										x								x
PETRUCHIO	x		x	x	x		x		x	x			x	x	x	x				x	x					x	x	x	x			x	x		x
PNEditor					x				x											x						x									x
Yasper			x	x	x				x				x	x			x			x	x											x			x
PAPETRI	x				x				x				x								x													x	x
Xpetri			x		x				x				x									x						x						x	x
PROD	x				x					x	x							x									x	x	x	x		x			x
ARP				x	x					x				x	x	x					x	x									x				x
JPetriNet				x	x				x													x				x									x
Petri .NET Simulator				x	x				x				x	x							x											x			x
QPME	x		x		x			x	x					x						x			x			x	x	x	x			x	x		x

Table 1: Comparison of twenty Petri Net Tools

Table 1 shows twenty Petri Nets tools compared in terms of 5 main categories which is Petri Nets supported, components in the tool, analysis type available, environments to run in and availability of the tool. The first group of comparison is the Petri net supported. In this category, the Petri net tools are compared in terms of what kind of Petri net is supported. Most of the tools support Place/Transition Petri net with some supporting high-level Petri net (i.e. AlPiNA, CoopnBuilder, PROD, and QPME). However, QPME stands out in this category as it supports Queuing Petri net (a combination of Queuing Network and Petri net). For Continuous Petri net, only Snoopy and Charlie support it.

The second categories compared are the components available in each tool. Majority of the tools provide a graphic editor and a fast simulation on Petri net. Tools that provide graphic editor and fast simulation on Petri net can be good education tools. Users will be able to create, edit Petri net and simulate different Petri net while learning about Petri net. Amongst all the tools compared, PEP has the most number of components which includes graphical editor, state spaces, condensed state spaces, token game animation, place invariants, transition invariants, net reduction, model checking, Petri net generator and interchange file format. Users will be able to explore more about Petri Nets while using PEP compared to the other tools.

The next form of comparison is the types of analysis for Petri net. Some of the tools surveyed deliver simple performance analysis, while tools such as GreatSPN, PEP, Charlie, JSARP, Xpetri, ARP and JPetriNet offer structural analysis. GreatSPN and QPME are also able to carry out advance performance analysis. The reachability graph based analysis is however only able to be performed by MARCIE.

The next sort of comparison is the environment types that the Petri net tools support. LoLA, PETRUCHIO and QPME have the highest amount of environment types supported for their tools with six environments for each of them including Windows, Macintosh and MS Dos. However, tools like AlPiNA, CoopnBuilder, MIST, Yasper, PAPETRI, ARP and Petri .NET Simulator is very environment specific with each of them only supporting one specific environment to be run on.

The final criterion of comparison is the pricing of the Petri net tools. All of the Petri net tools are either free of charge or free of charge for academic purpose to be downloaded.

This thesis aims to create an algorithm to map Petri Nets to Sequence Diagram. The algorithm created might be similar to the functions of the Petri Net tools surveyed above as code generation is also one of the many functions of the Petri Net tools. The survey of Petri Net tools will also benefit the users in choosing which Petri Net tools to be used in analyzing Petri Nets.

2.2 Unified Modeling Language (UML)

Unified Modeling Language (UML) is an informal modeling language used to offer a standard and unified way to visualize the design of a system. [12]. It has been accepted as a standard by the Object Management Group (OMG). UML are used to show the structural and behavioral view of the design in a system. Many UML tools have been created to carry out different function such as diagramming, round-trip engineering, code generation, reverse engineering, model transformation and model and diagram interchange. As of August 2014, there are 19 registered OMG members for UML vendor, 43 nonmembers UML vendor and a lot more UML tools that are not registered under OMG [13].

There have been numerous versions of the UML, such as UML 1.0, UML 1.4 and UML 1.5 which were all merely minor revisions of UML 1.0. UML 2.0 is a major revision for the UML 1.5 version. To date the latest formal UML version is the UML 2.4.1 [14].

UML can model the system in two types of diagrams, which is structural and behavioral diagram. Structural diagram emphasizes on the fixed structure of the system using objects, operations and relationships. Examples of structural diagrams include class diagrams and composite structure diagrams. Behavioral diagram shows the dynamic behavior of the system. The behavioral view shows the relationship and interactions between the objects in the system. Examples of behavioral diagrams include sequence diagrams, activity diagrams and state machine diagrams.

UML 2.0 consists of different types of diagrams which are divided into two main categories, one is the structural diagram and the other one is the behavior diagram. The figure below illustrates how UML 2.0 is categorized.

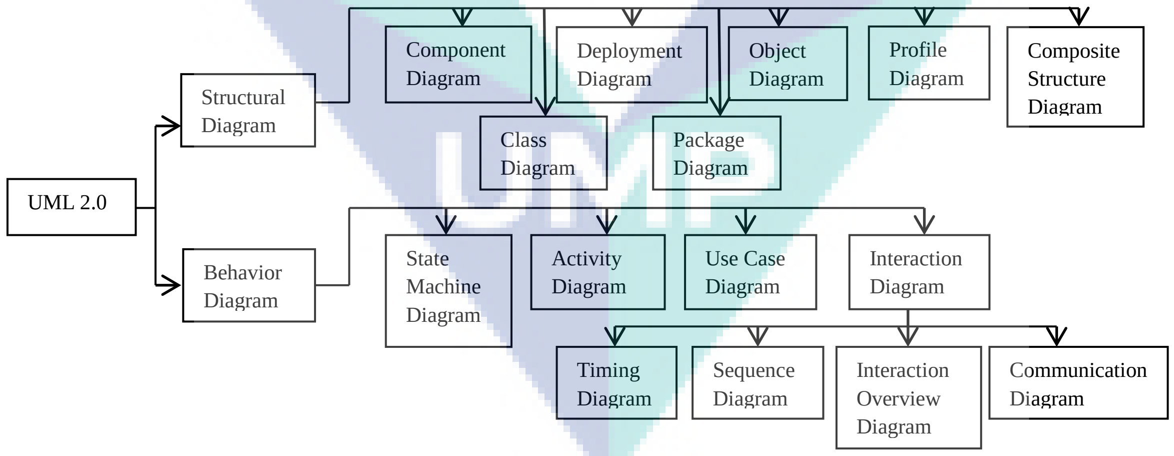


Figure 4: UML 2.0 Hierarchy

Different diagrams provide different type of perspectives to the developer. For example, by analyzing a class diagram (structural diagram), developer can focus solely on the system's classes, attributes, operations and the relationship between the objects. While analyzing a sequence diagram (behavior diagram), developer can focus on how the object interacts with each other and the sequence of the processes in the system. UML enables software engineers to break down a system into different diagrams which describes the system from different perspectives.

2.2.1 SEQUENCE DIAGRAM

Sequence diagram is an interaction diagram that shows how processes work with each other and in what order. It is a construct of a message sequence chart [15]. It illustrates object interactions arranged in time sequence. Components in a sequence diagram are lifeline, message, interaction operator, event and combined fragment. The figure below is an example of sequence diagram,

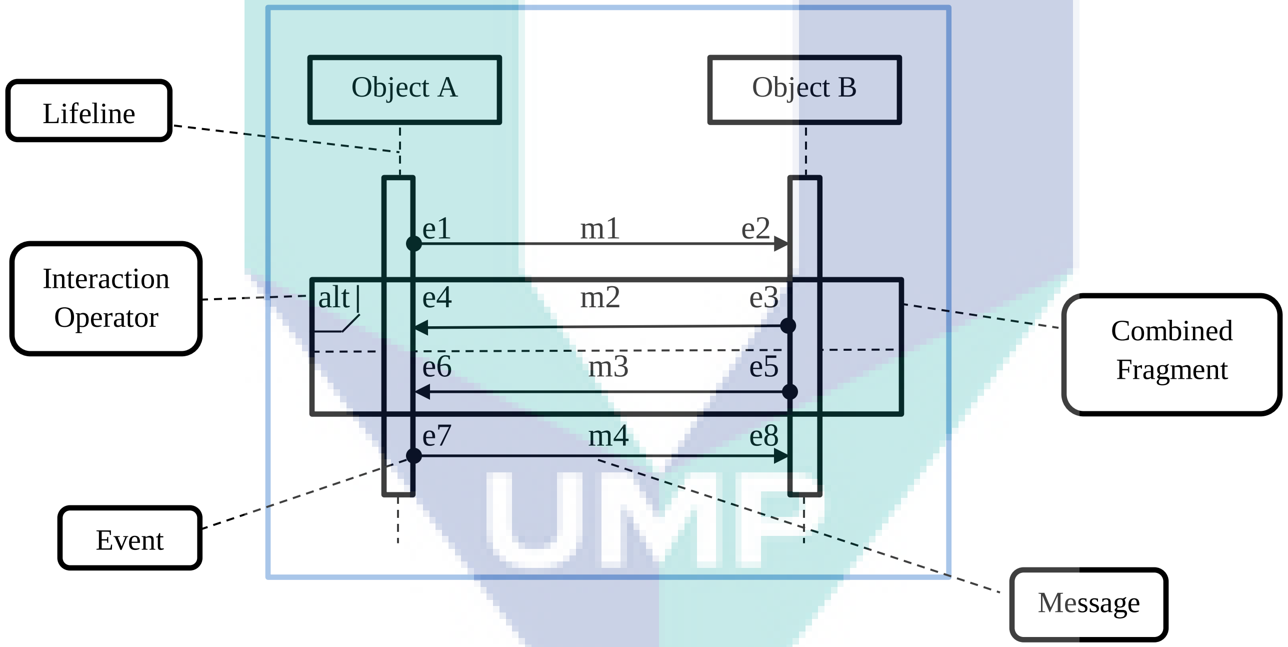


Figure 5: Example of Sequence Diagram

Figure 5 is an example of Sequence Diagram with two lifelines (objects) and four messages. The messages indicates communication between Object A and Object B. Message m1 and m4 shows interaction from Object A to Object B, while message m2 and m3 shows interaction from Object B to Object A. Lifelines are vertical lines that represent objects or an instance of a class in a system while messages are horizontal arrows that begins and ends at a lifetime. These messages denote the communication among the objects that are represented by the particular lifelines. Messages are normally used as a sign or a call for procedure or function in the system.

Figure 5 also shows a concept of events labelled as e1, e2 … e8. In a normal Sequence Diagram, events are not labelled. The label in Figure 5 is done deliberately so that it can familiarize readers with the concept of events used and described later on in this thesis. The mapping of Petri Nets to Sequence Diagram will be based on the concept of events. Events are attached to the lifelines and represent the sending and receiving of messages. Referring to [16], there are two main rules in the sequencing of events in a Sequence Diagram:

1. The events on each lifeline must be ordered from top to bottom.
2. The event that denotes the sending of a message must occur before the event that denotes the receiving of the same message.

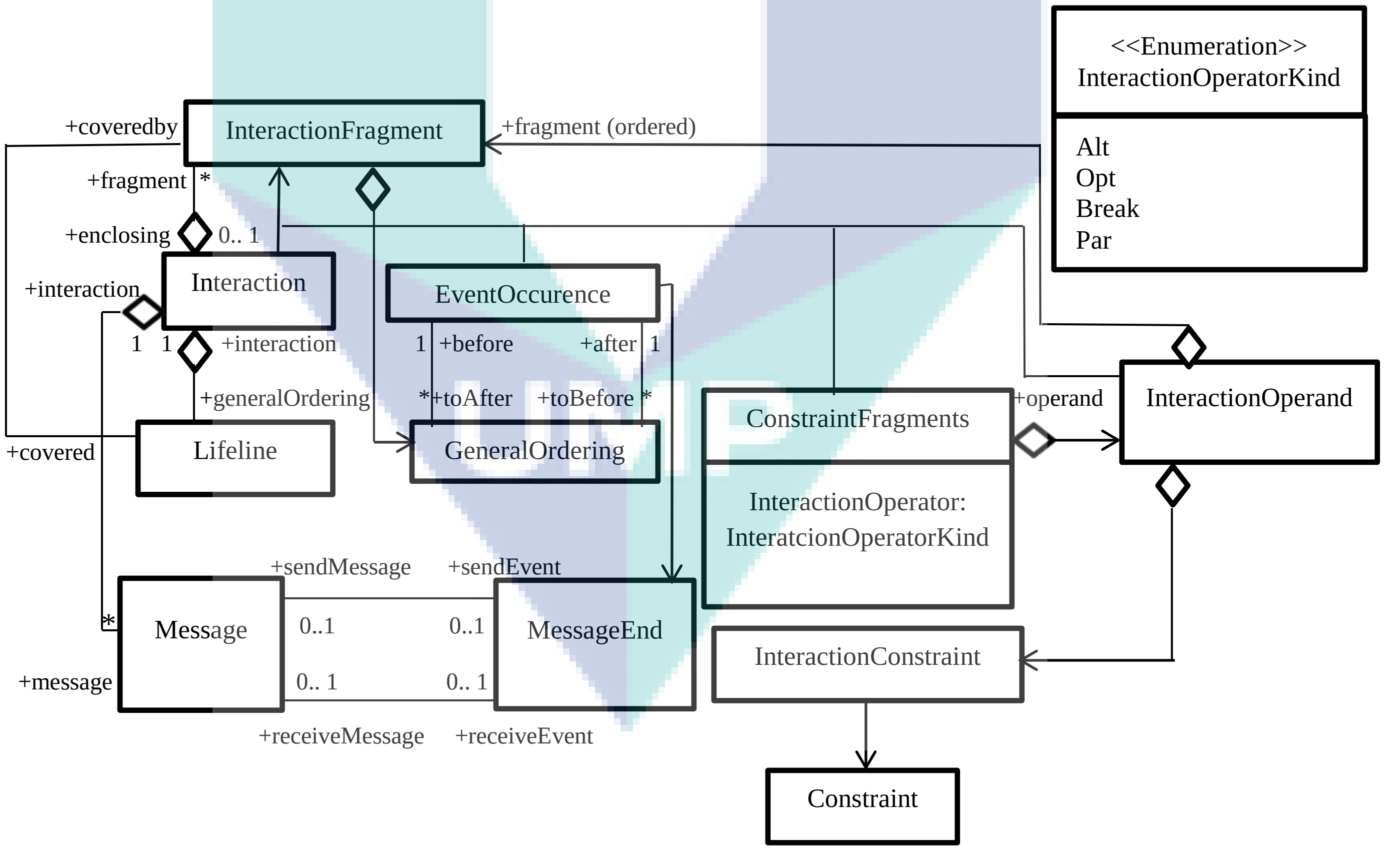


Figure 6: Sequence Diagram Metamodel

Figure 6 illustrates the metamodel for UML Sequence Diagram. The main components of a Sequence Diagram are lifelines, messages and Combined Fragments that are defined by Interaction Operators. Combined Fragments are high level additions introduced to Sequence Diagrams. A Combined Fragment is defined by the Interaction Operator that is attached to it, as well as the amount of operands it has. As shown in Figure 5, a Combined Fragment with the Interaction Operator *alt* is presented with two operands. The numbers of Operands are determined by the number of fragments in the Combined Fragment. Interaction Operators are used as a mechanism to provide structure in the communication between lifelines. In Figure 5, the Interaction Operator *alt* refers to *alternative* (conflicting) behavior where only messages in one of the two operands would be executed. Hence, if the message m2 is sent, then the message in the second operand, m3 would not be sent or vice versa. There are eleven types of Interaction Operators as shown by [17]; however only four types of the Interaction Operators will be used in this thesis, which will be introduced in the following Table 2.

Table 2: Types of Interaction Operator

Interaction Operator	Abbreviation	Semantics Description
Alternative	alt	Alternative Interaction Operator is a choice of behavior where at most only one of the operands in the Combined Fragment is chosen. The operands of the Combined Fragment could be assigned a guard or constraint that has to be evaluated to be true for it to be chosen.
Option	opt	The Option Interaction Operator is similar to the Alternative Interaction Operator, which is a choice of behavior occurs. The default for an option Interaction Operator is one operand, where either the operant happens, or nothing happens.
Break	-	A Break Interaction Operator is a choice of behavior where an operant occurs, or the remainder of the interaction is disregarded (i.e. the termination of the system). The operands could be attached to a guard to determine the chosen behavior. However, a Break Interaction Operator without a guard leads to a non-deterministic choice of behavior.
Parallel	par	The Parallel Interaction Operator shows that a parallel merge between all the operands of the Combined Fragment occur. The order of the messages within each operand of the Combined Fragment is preserved. However, the order of the messages between operands can be interleaved in

		any variations.

2.2.2 UML Tool

A UML tool is a tool that supports some or all of the semantics associated with UML, such as the structural diagram or behavioral diagram. There are two sorts of UML tools, either a standalone software tool or a plugin tool for existing software.

Basically, UML tools can be characterized based on their functionality. The figure below shows the categories of UML tools.

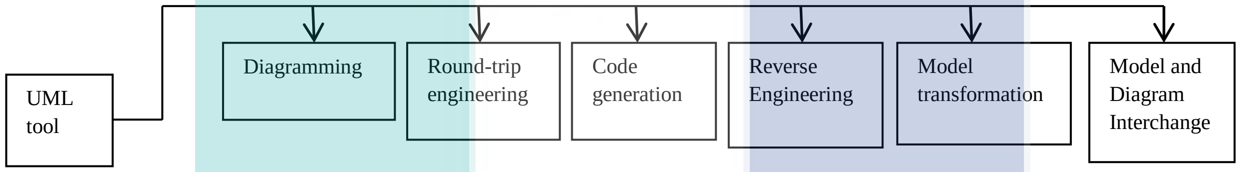


Figure 7: Types of UML Tools

UML tools based on diagramming function are used to create and edit UML diagrams. Developers can use diagramming tools to draw diagrams of object-oriented software as long as UML notations are followed. UMLet, ArgoUML and Visio are some examples of UML diagramming tools.

A round-trip engineering UML tool is able to perform code generation from models and also model generation from code, while keeping both the model and the code semantically consistent. A few examples of round-trip engineering UML tools are Altova UModel and UML Lab.

In code generation UML tool, UML diagrams are used to generate codes. The code generation function will provide a rough structural code in response to the UML diagrams provided. This in turn will benefit the programmer as one do not need to code from scratch. Examples of code generation UML tools are Acceleo and AthTek Flowchart to Code.

For reverse engineering UML tools, the UML tool reads source code as an input and creates corresponding UML diagrams based on it. An example for reverse engineering UML tool is the Architexa.

In model and diagram interchange UML tool, UML models are represented by XML Metadata Interchange (XMI). XMI is not supported by UML diagram interchange; hence it allows the importation of UML diagrams from one model to another. Examples of UML tools that supports model and diagram interchange are Poseidon, Adobe SVG plugin and Batik.

For model transformation UML tool, the concept is to associate it with model-driven architecture. Hence, the tool is capable of transforming a model into another model. Examples of model transformation UML tools are UMT-QVT and UML RSDS.

A survey on UML tools [18] was carried out recently with the tabulated result as of below. The reason for carrying out the survey is to have an updated list of comparison between the latest versions of ten UML tools

Table 3: A survey of UML tools

UML Tool			**ArgoUML**	**Modelio**	**UModel**	**Visual Paradigm for UML**	**Rational Software**	**Software Ideas Modeler**	**Umbrello UML Modeler**	**UMLet**	**BOUML**	**Papyrus**
Comparison	**Latest supported version**	**UML 1.4**	x							x		
		UML 2.0							x		x	
		UML 2.4		x	x	x	x	x				x
	Diagrams supported	**Structural**	x	x	x	x	x	x	x	x	x	x
		Behavior	x	x	x	x	x	x	x	x	x	x
	Model Driven Architecture		x	x	x	x	x	x	x		x	x
	XML Metadata Interchange		x	x	x	x	x	x	x		x	x
	Languages code generated	**C#**	x	x	x	x	x	x	x			
		C++	x	x		x	x	x	x		x	x
		JAVA	x	x	x	x	x	x	x		x	x
		PHP	x			x		x	x		x	
		Visual Basic			x							
		VB.Net				x		x				

UML Tool			ArgoUML	Modelio	UModel	Visual Paradigm for UML	Rational Software	Software Ideas Modeler	Umbrello UML Modeler	UMLet	BOUML	Papyrus
Comparison	Languages code generated	SQL		x			x	x	x		x	
		C#		x	x	x		x	x			
		C++		x		x	x		x		x	
		JAVA	x	x	x	x	x	x	x		x	x
		PHP				x		x			x	
		Visual Basic			x							
		VB.Net				x		x				
	Type	Standalone tool	x	x	x	x	x	x	x	x	x	x
		Plugin/ Integration			x	x	x		x	x		x
	Price	Free	x	x		x		x	x	x		x
		Paid		x	x	x	x	x			x	
	Platform	Windows	x	x	x	x	x	x	x	x	x	x
		Linux	x	x		x	x	x	x	x	x	x
		Mac OS X	x	x		x	x		x	x	x	x

The table above compares ten UML tools in terms of the latest versions of UML supported, diagrams supported, model driven architecture supported, XML Metadata Interchange supported, languages code generated, languages reverse engineered, type of tools, price and the type of platform.

The first type of comparison is the version of UML supported. In this part, ArgoUML, Modelio, Visual Paradigm for UML, Rational Software Architect, Software Ideas Modeler and Papyrus has the best support for providing the latest UML 2.4 standard to the users. In terms of diagrams supported, all the UML tools surveyed support both structural and behavior diagrams. The tools also support model driven architecture and XML Metadata Interchange except for

UMLet. Nine out of the ten UML tools surveyed supports JAVA programming languages in code generation and reverse engineering. All of the surveyed UML tools can also work as a standalone tool while UModel, Visual Paradigm for UML, Rational Software Architect, Umbrello UML Modeler, UMLet and Papyrus provides plugin or integration with another IDE tool such as Netbean or Eclipse. Most of the UML tools surveyed have free version to be used non-commercially, while the rest have free trial versions which will expire after a few months of usage. For some of the UML tools to be fully utilized, users are recommended to purchase the license to unlock the full function of the UML tool. Most of the UML tools can be installed in cross platform (Windows, Linux and Mac OS X) as long as that particular platform supports Java.

Based on the survey [18], UML tools that are free of charge will be used in this research. After transforming Petri Nets back to Sequence Diagram, users will be able to choose which UML tool to be used in editing the Sequence Diagram created.

2.3 SD2PN

SD2PN is a tool used to perform model transformations from Sequence Diagram to Petri Nets [19]. It also functions as a framework for Sequence Diagrams to be transformed into Petri Nets. The model transformation from Sequence Diagram is broken down into three stages to illustrate the stages involved in the process; they are *Decomposition*, *Transformation* and *Composition*. The three stages are further explain as below,

Stage 1 (*Decomposition*) : Decomposing the Sequence Diagrams into fragments. The Sequence Diagram inputted into SD2PN is decomposed into several small fragments based on the Sequence Diagram metamodel.

Stage 2 (*Transformation*) : Transforming each fragment of Sequence Diagrams into blocks of Petri Nets. Sequence Diagram fragment obtained from Step 1 is transformed into a Petri Net block respectively based on a set of model transformation rule.

Stage 3 (*Composition*) : The Composition stage consists of two functions, which is morphing and substituting. In this step, the blocks of Petri Nets are morphed and substituted to create a Petri Net representation of the original Sequence Diagram.

2.3.1 DECOMPOSITION

In this decomposition stage, based on [19], a message can be referred to an event, or the flow of information between the objects in Sequence Diagrams, so a message is considered to be a Sequence Diagram fragment. The four InteractionOperatorKind (*alternative*, *option*, *break* and *parallel*) as shown in Figure 6 are able to change the flow of events in a different way, hence they each are designated as a fragment type too. In short, there are five types of Sequence Diagram fragments, which are *message*, *alternative*, *option*, *break* and *parallel*. The Sequence Diagram inputted into SD2PN will be decomposed based on these five types of fragments. The decomposition still preserves the causality of the messages or the hierarchical structure of the CombinedFragments.

2.3.2 TRANSFORMATION

In this stage, Sequence Diagrams fragments will be transformed into Petri Net blocks. In the transformation stage, five transformation rules must be followed based on each of the five types of fragments as shown in the decomposition stage.

The concept of placeholders and Petri Net blocks are also introduced into the extended Petri Nets metamodel before the transformation process. Placeholders are temporary nodes that mimic the structure of a place in Petri Nets. Petri Net blocks are blocks of Petri Nets that have unique input and output places, which are referred to as precondition and postcondition separately.

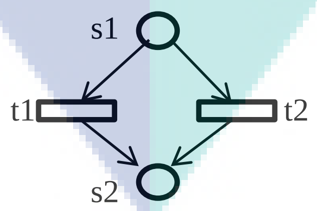


Figure 8: Example of a Petri Net block

The figure above shows an example of a Petri Net block. Petri Net blocks can also be expressed formally as of below.

A Petri Net block is a four tuple $B = (S, T, P, F)$ where S is a finite set of *places*, T is a finite set of *transitions*, and P is a finite set of *placeholders*. $F \subseteq ((S \cup P) \times T) \cup (T \times (S \cup P))$ is a set of *arcs*. *In*(B), *Out*(B) ∈ S are *unique* places (*precondition* and *postcondition* respectively) such that *In*(B) has no incoming arcs and *Out*(B) has no outgoing arcs. They symbolize the start and end places in the Petri Net blocks correspondingly. Petri Net bock can also be textually represented as the sum of all its components. As an example, the Petri Net block in Figure 8 can be written as

$$B=(\{s1,s2\},\{t1,t2\},\{\ \},\{(s1,t1),(s1,t2),(t1,s2),(t2,s2)\}).$$

For larger Petri Net blocks where the textual representation such as above may be complicated, hence it may also be written as $B = (S, T, P, F)$ where

$$S = \{s1, s2\}$$

$$T = \{t1, t2\}$$

$$P=\{\ \}$$

$$F = \{(s1, t1), (s1, t2), (t1, s2), (t2, s2)\}$$

With placeholders and Petri Net blocks introduced, Sequence Diagram fragments can be transformed using a set of SD2PN transformation rules which will be explained in the following section.

2.3.2.1 SD2PN Rule 1: Transforming Message fragments

SD2PN Rule 1, the model transformation for *message* fragments. The execution of a message, *m* in a Sequence Diagram is represented as the firing of a transition in the corresponding Petri Net. For each *message* fragment that exists in the Sequence Diagram, a Petri Net block is created. This transformation rule result in a Petri Net block textually as

$$B = (\{s1, s2\},\{m\},\{\ \},\{(s1, m),(m, s2)\})$$

Figure 9 below illustrates how SD2PN Rule 1 is applied when converting a message fragment from Sequence Diagram to Petri Net block.

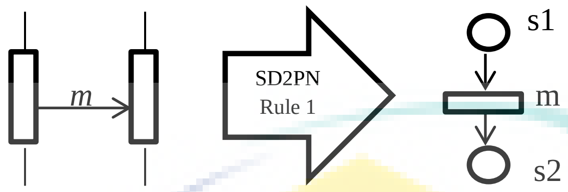


Figure 9: Applying SD2PN to message fragment

For each *message* fragment that exists in Sequence Diagram, a Petri Net block is created. The Petri Net block is made up of two places, s1 and s2. These places indicate the precondition and postcondition of the Petri Net block individually. The *message*, m in the Sequence Diagram fragment is transformed into a transition in the Petri Net block and labelled with the same name.

2.3.2.2 SD2PN Rule 2: Transforming Alternative Fragments

SD2PN Rule 2 is the model transformation rule for *alternative* fragments. The Interaction Operator *alternative* specifies that a set of event may occur if a condition is satisfied and another set of event will occur if otherwise. For each CombinedFragment with the InteractionOperatorKind *alternative* in the Sequence Diagram, a Petri Net block is created. This results in a Petri Net block written as $B = (S, T, P, F)$ where

$$S = \{s1, s2\}$$

$$T = \{t1, t2, t3, t4\}$$

$$P = \{ph1, ph2\}$$

$$F = \{(s1, t1),(s1, t2),(t1, ph1), (t2, ph2), (ph1, t3), (ph2, t4), (t3, s2), (t4, s2)\}$$

Figure 10 below shows how SD2PN Rule 2 is applied to an alternative fragment when converting a Sequence Diagram fragment to a Petri Net block.

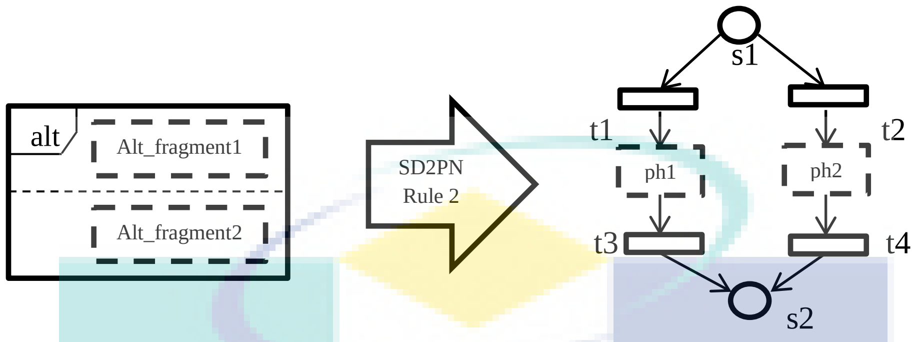


Figure 10: Applying SD2PN to alternative fragment

For each CombinedFragment with the InteractionOperatorKind *alternative* in the Sequence Diagram, a Petri Net block is created. The Petri Net block comprises of two places, s1 and s2 to model the precondition and postcondition. The Petri Net block also have two *placeholders* which are *ph1* and *ph2* that acts as temporary places that will be swapped by the events in the operand *alt_fragment1* and *alt_fragment*2. The behavior of the alternative fragment is indicated by two transitions *t1* and *t2* with incoming arcs from the precondition, only one of the two transitions may fire. Transition *t1* and *t2* are connected to *ph1* and *ph2* respectively. Then two more transition *t3* and *t4* are created to represent the end of the *alternative* fragment. Transition *t3* receives an incoming arc from *ph1* while *t4* receives incoming arc from *ph2*. Both *t3* and *t4* are connected via an outgoing arc to the postcondition.

2.3.2.3 SD2PN Rule 3: Transforming Option Fragments

SD2PN Rule 3 is the model transformation rule for *option* fragments. The Interaction Operator *option* is similar to the *alternative* fragment. For each CombinedFragment with the InteractionOperatorKind *option*, a Petri Net block is generated. The result is a Petri Net block written as $B = (S, T, P, F)$ where

$$S = \{\text{s1, s2}\}$$

$$T = \{\text{t1, t2, t3, t4}\}$$

$$P = \{\text{ph1, ph2}\}$$

$$F = \{\text{(s1, t1),(s1, t2),(t1, ph1), (t2, ph2), (ph1, t3), (ph2, t4),}$$

(t3, s2), (t4, s2)}

Figure 11 below illustrates how SD2PN Rule 3 is applied when converting an *option* fragment from Sequence Diagram to Petri Net.

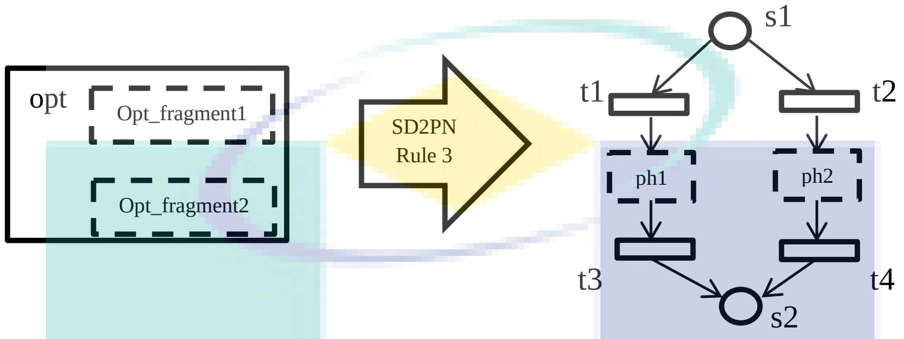


Figure 11: Applying SD2PN to option fragment

The difference between the generated Petri Net block is that a Combined Fragment of type *option* may contain just one operand. Since only one operand exists, there must only be one placeholder in the Petri Net block. Thus the placeholder in *ph2* is substituted with a place skip that mimics the system where the actions inside the *option* operand are 'skipped'. The resulting Petri Net block is as show in below,

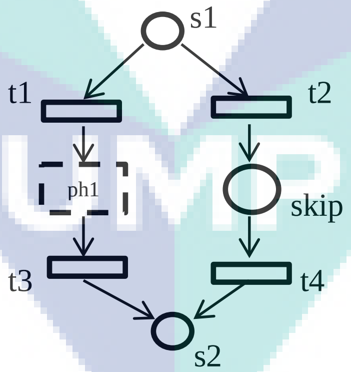


Figure 12: A Petri Net block with an option fragment with only one operand

Similar to Rule 2, if the CombinedFragment consists of the first message of the Sequence Diagram (including inside nested fragments), the precondition of the resulting Petri Net block must contain a token.

2.3.2.4 SD2PN Rule 4: Transforming Break Fragments

SD2PN Rule 4 is the model transformation rule for *break* fragment. The Interaction Operator *break* consist of a condition such that when it is satisfied, the operation *breaks* (terminates). It is a specialization of the 'if… else…' construct. Each *break* fragment is transformed into a corresponding Petri Net block. To show termination of the system, a *place* marked with X is created, which is also known as a terminal node. The resulting Petri Net block can be written as $B = (S, T, P, F)$ where

$$S = \{s1, s2, X\}$$

$$T = \{t1, t2, t3\}$$

$$P = \{ph\}$$

$$F = \{(s1, t1),(s1, t2),(t1, ph),(t2, X),(ph, t3),(t3, s2)\}$$

Figure 13 below shows how SD2PN Rule 4 is applied when converting a break fragment of Sequence Diagram to Petri Net.

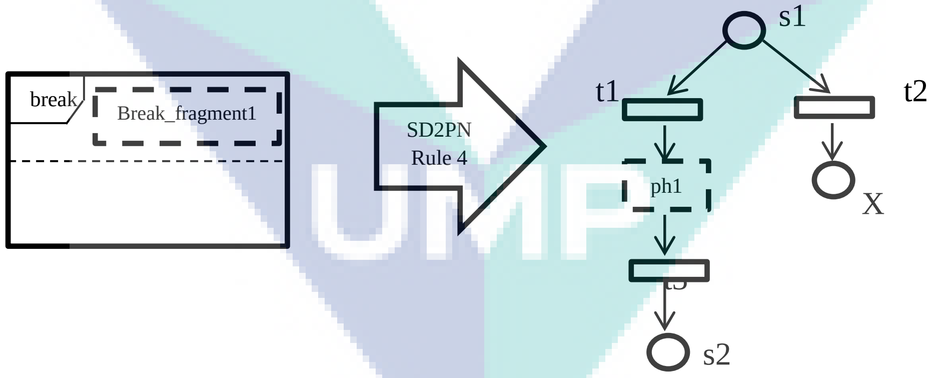


Figure 13: Applying SD2PN to a break fragment

A Petri Net block is created for every CombinedFragment of type break that exists in the Sequence Diagam. This Petri Net block contains precondition and postcondition modelled as places *s1* and *s2*. The operand inside break fragment is modelled by a placeholder in the Petri Net block. Alike the rules before this, two transitions *t1* and *t3* are used to connect the placeholder to the precondition and postcondition. A place marked by X is created to show the termination of the system which is also known as a terminal node. The terminal node is connected to the precondition via transition *t2*. But it is not connected to the postcondition as the system is terminated at X.

2.3.2.5 SD2PN Rule 5: Transforming Parallel Fragments

SD2PN Rule 5 is the model transformation rule for *parallel* fragment. The Interaction Operator *parallel* specifies two or more sets of event should occur concurrently without any pre-defined set of conditions. There should not be any causality or conflicting event between all the operands of the *parallel* fragment. For each Combined Fragment of type *parallel* that exists in the Sequence Diagram, a Petri Net block is generated. The resulting Petri Net block can be written as $B = (S, T, P, F)$ where

$$S = \{\mathrm{s1}, \mathrm{s2}\}$$

$$T = \{\mathrm{t1}, \mathrm{t2}\}$$

$$P = \{\mathrm{ph1}, \mathrm{ph2}\}$$

$$F = \{(\mathrm{s1}, \mathrm{t1}),(\mathrm{t1}, \mathrm{ph1}),(\mathrm{t1}, \mathrm{ph2}),(\mathrm{ph1}, \mathrm{t2}),(\mathrm{ph2}, \mathrm{t2}),(\mathrm{ts}, \mathrm{s2})\}$$

Figure 14 below illustrates how SD2PN Rule 5 is applied when converting a parallel fragment of Sequence Diagram to Petri Net.

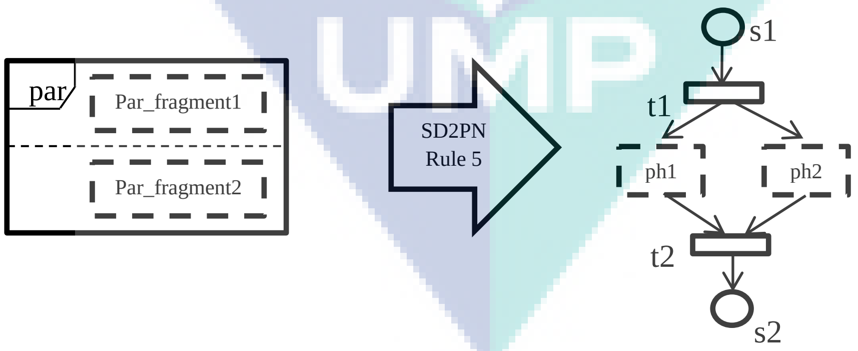


Figure 14: Applying SD2PN to a parallel fragment

The resulting Petri Net block consists of a precondition (*s1*), postcondition (*s2*) and placeholders *ph1* and *ph2* that model the operands par_fragment 1 and par_fragment 2. To model the concurrency between the operands, a single *transition t1* is used to connect all the placeholders to the precondition. This is due to the firing of *t1* will provide tokens to the both *ph1* and *ph2* in parallel. Transition *t2* is then created to connect the placeholders to the postconditions.

The description above briefly describes how does SD2PN works when applying it to a Sequence Diagram to transform it into Petri Net. The correctness of the model transformation performed by SD2PN has been proven mathematically using a common semantics domain in Labelled-Event Structures [20] by using the semantic mapping introduced by Kuster-Filipe, J and McMillan, K.L in their respective works [21].

2.3.3 COMPOSITION

After mapping each Sequence Diagram fragment into Petri Net blocks, the Petri Net blocks need to be composed into an integrated Petri net that resembles to the original Sequence Diagram. Based on an observation, there is a commonality between all the Petri Net blocks created via SD2PN, which is each of the Petri Net blocks have a single input and output place, also known as precondition and postcondition. This is done purposely to allow a constant method of putting the Petri Net blocks back together. In this composition stage, there are two local functions used, which is *morph* and *substitute*.

2.3.3.1 MORPH

The *morph* function is used to combine causal Petri net blocks. In formal descriptions, the symbol ⊗ is used to indicate the *morph* function. Hence the morphing of B1 and B2 can be represented formally as $B_1 \otimes B_2$. The causality relationship is derived from the *GeneralOrdering* from the Sequence Diagram metamodel in Figure 6 in section 2.2.1 of this thesis. The *morph* function is used to attach Petri Net blocks by merging the *postcondition* of a block with the *precondition* of another block, which enforce a causal behavior. Hence the *morph* function can only be called when there are two Petri Net blocks at a time.

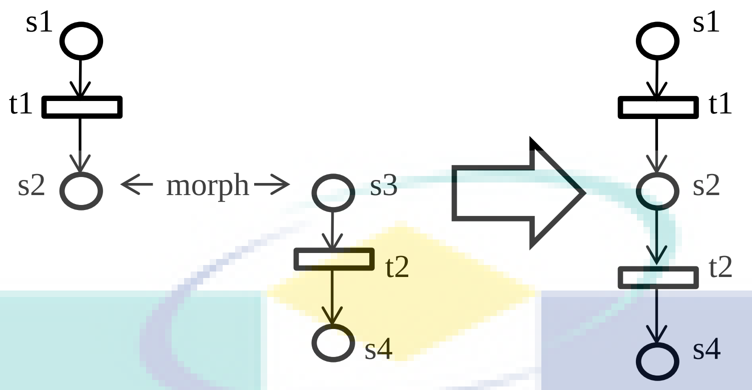


Figure 15: Example of morph function between two Petri Net blocks

Figure 15 shows the theory of *morphing* between two Petri Net blocks. When the *morph* functions is called on the two Petri Net blocks, the *postcondition* of the former is combined with the *precondition* of the latter, creating an integrated Petri Net block. Based on the example above, the *morphed* place will always take the label of the former block and also ignoring the latter block. In this case, after the morph process, s2 is chosen to label the place while s3 label is ignored. It can also be written formally as of below.

Suppose $B_1 = (S_1, T_1, P_1, F_1)$ and $B_2 = (S_2, T_2, P_2, F_2)$ are two Petri Net blocks. The *morphing* of B_1 and B_2 is represented by $B_1 \otimes B_2$ which results in a new Petri Net block $B = (S, T, P, F)$ such that $T = T_1 \cup T_2$, $P = P_1 \cup P_2$, $S = (S_1 \cup S_2) \setminus \{Out(B_1)\}$, $In(B) = In(B_1)$ and $Out(B) = Out(B_2)$ and

$$F = ((F_1 \cup F_2) \setminus \{(x,y) \mid y = Out(B_1)\} \cup \{(x, In(B_2) \mid (x, Out(B_1) \in F_1\}\ldots\ldots(*).$$

To explain about (*), notice that the arcs in B are acquired by including all the arcs in $F_1 \cup F_2$ excluding the arcs leading to output places of B_1,$Out(B_1)$. All arcs that terminates in $Out(B_1)$ must be redirected to $In(B_2)$ in order to *morph* B_1 to B_2.

Based on the example in Figure 15, suppose that the two Petri Net blocks B_1 and B_2 such that B_1 = ({s1, s2}, {t2}, { }, {(s1, t1), (t1, s2)}) and B_2 = ({s3, s4}, {t2}, { }, {(s3, t2), (t2, s4)}) where $s1$ and $s3$ are preconditions of B_1 and B_2 respectively while $s2$ and $s4$ are postconditions of B_1 and B_2. Triggering the morph function as $B_1 \otimes B_2$ will combine the postcondition of B_1 and the precondition of B_2, creating a Petri Net block represented formally as of below.

$$B = (\{s1, s2, s4\}, \{t1, t2\}, \{\ \}, \{(s1, t1), (t1, s2), (s2, t2), (t2, s4)\})$$

2.3.3.2 SUBSTITUTE

Substitute is a function used for composing hierarchical behavior between Petri Net blocks. The function is used only to replace a *placeholder* with a Petri Net block. The *substitute* function is called repeatedly until there are no more *placeholders*. The function can also be written mathematically as $B_2[B_1/p]$, which means a *placeholder p* inside B_2 is replaced with B_1.

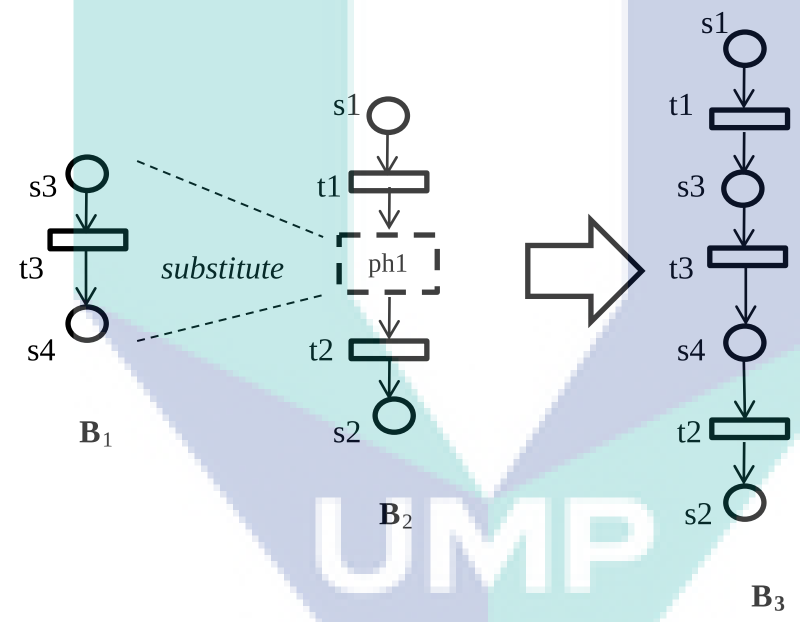


Figure 16: Example of substitute function between Petri Net blocks

Figure 16 shows an example of *substitution* between two Petri Net blocks and the result of the *substitution* process. Each time the *substitute* function is triggered, a *placeholder* (from B_2) is substituted by another Petri Net block (B_1) such that the incoming arc into the *placeholder* (from B_2) is transferred into the *precondition* of the block (B_1), while the outgoing arc from the *placeholder* (from B_2) is transferred as if from the *postcondition* of the block (B_1). This can also be represented formally as of below.

Suppose $B_1 = (S_1, T_1, P_1, F_1)$ and $B_2 = (S_2, T_2, P_2, F_2)$ are two Petri Net blocks. Let *ph1* be a *placeholder* in B_2. The *substituting* of the Petri Net block, B_1 into *ph1* is represented by the notation $B_2[B_1/ph1]$ which results in a Petri Net block, $B = (S, T, P, F)$, where

$$S = S_1 \cup S_2,\ T = T_1 \cup T_2,\ P = (P_1 \cup P_2) \setminus \{ph1\},\ In(B) = In(B_2),\ Out(B) = Out(B_2)$$

and

$$F = (F_1 \cup F_2 \setminus \{(x, y) \mid x = ph1 \text{ or } y = ph1\}) \cup \{(x, In(B_1)) \mid (x, ph1) \in F_1\} \cup \{(Out(B_1), y) \mid (ph1, y) \in F_1\} \ldots\ldots (**).$$

The equation (**) means that arcs in B can be acquired by removing all arcs to and from *ph1* and redirecting them to $In(B_1)$ and $Out(B_2)$ respectively. Another example is presented in Figure 17 below since most cases of substitution in SD2PN involves the necessity for two Petri Nets to be substituted into one Petri Net block with two placeholders (alternative, parallel and some cases in option fragments).

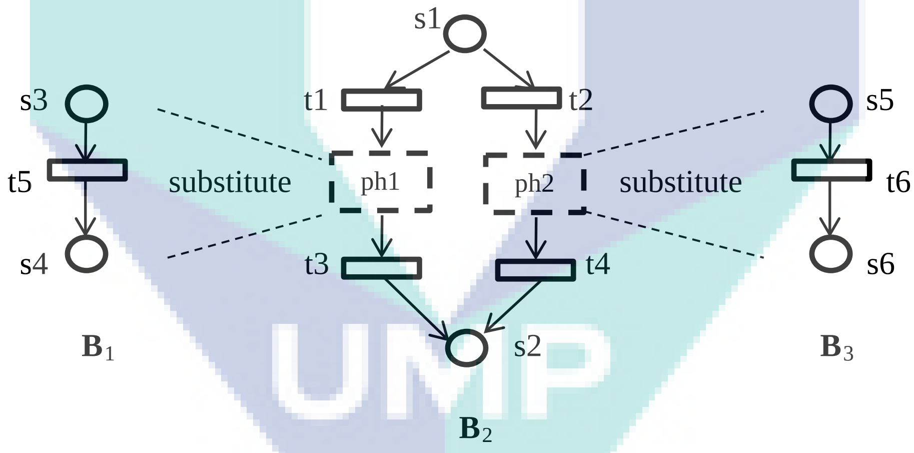


Figure 17: Example of two *substitute* actions between Petri Net blocks

B_1, B_2 and B_3 can be represented formally as of below.

$$B_1 = (\{s3, s4\}, \{t5\}, \{\ \}, \{(s3, t5), (t5, s4)\})$$

$$B_2 = \left(\begin{array}{c} \{s1, s2\}, \{t1, t2, t3, t4\}, \{ph1, ph2\}, \\ \left\{ \begin{array}{c} (s1, t1), (s1, t2), (t1, ph1), \\ (t2, ph2), (ph1, t3), (ph2, t4), (t3, s2), (t4, s2) \end{array} \right\} \end{array} \right)$$

$$B_3 = (\{s5, s6\}, \{t6\}, \{\ \}, \{(s5, t6), (t6, s6)\})$$

By triggering the substitute function $B_2[B_1/ph1]$ and $B_2[B_3/ph2]$, it results in a Petri Net block, $B = (S, T, P, F)$ where

$$S = \{s1, s2, s3, s4, s5, s6\}$$

$$T = \{t1, t2, t3, t4, t5, t6\}$$

$$P = \{\ \}$$

$$F \quad \left\{ \begin{array}{l} (s1, t1), (s1, t2), (t2, s3), (s3, t5), (t5, s4), (t2, s5), \\ (s5, t6), (t6, s6), (s4, t3), (s6, t4), (t3, s2), (t4, s2) \end{array} \right\}$$

Figure 17 illustrates the case where the *substitution* is between three Petri Net blocks such that B_1 and B_3 are substituted into B_2. The order by which the substitution takes place is arbitrary. The substitution of $B_2[B_1/ph1]$ followed by $B_2[B_3/ph2]$ produces the same result as the substitution of $B_2[B_3/ph2]$ followed by $B_2[B_1/ph1]$. Hence,

$$B_2[B_3/ph2][B_1/ph1] = B_2[B_1/ph1][B_3/ph2]$$

2.4 Similar Works

In this section, other model transformation tools will be compared and reviewed.

2.4.1 UML2Alloy

UML2Alloy is a tool similar to SD2PN which attempts to bridge the gap between model design and model analysis [22]. The tool is the result of a research which tries to formalize the UML with the aid of Alloy (a declarative specification language for expressing complex structural constraints and behavior in a software system) via the concept of Model Driven Architecture [23]. In UML2Alloy, UML Class Diagrams are augmented with OCL constraints (a textual notation used to enforce constraints over UML model) and then transformed into Alloy models via Model Driven Development model transformation. The Alloy models created can then be analyzed using Alloy Analyzer which is a tool that enables model level analysis using first order logic. With UML2Alloy, users are able to benefit from the two critical functionalities of the Alloy Analyzer, which is *simulation* and *verification*. *Simulation* function makes sure that the model is not inconsistent while the *verification* function allows modelers to reason that certain critical properties of the model are satisfied.

UML2Alloy was developed on the platform of Java using SiTra [24] as the model transformation framework. Alloy is very suited to model static models and constraints; however it has certain limitations when it comes to dynamic behavioral models. Some dynamic properties could be modelled using *pre* and *post* conditions. Alloy does not have the mechanism to model complex behaviors such as parallelism.

2.4.2 UML-B/U2B

Another similar work is the UML-B which was done by Snook and Butler [25]. Via UML-B, the authors intended to offer a 'UML-like' graphical front end for B language, a formal language. It provides various diagrammatic modelling notations and editors for creating models which are then translated into B language for verification. The two notations (UML and B language) complement each other very well. The UML offers an accessible visualization of models facilitating communication of ideas but lacks of formal and precise semantics, while B language has the precision to support animation and rigorous verification. However many software engineers find the notation difficult to learn, visualize and communicate. In short, UML-B defines a formal modelling notation based on UML and some features borrowed from B language.

With the UML-B notation, U2B translator (a transformation tool) can be used to convert a UML-B model into its equivalent B specification [26]. The current version of U2B is a Rational Rose script.

2.4.3 Bridging the gap between Design and Analysis

From the tools above (UML2Alloy and U2B), it is noticeable that efforts are being done in bridging the gap between informal modelling language (UML) and formal modelling language (Alloy and B language). Informal modelling languages usually are used to design the system whilst formal modelling languages are usually used to perform analysis. Two different types of models created using two different sets of tools and using two different languages is what is described as model heterogeneity. In this case, heterogeneity exists between the two formal and informal modelling languages.

Heterogeneous models cannot communicate with each other under normal circumstances and requires a completely different skill-set to design. A software designer might be expert in informal modelling language like UML Sequence Diagram but not necessarily familiar with other formal modelling languages such as Alloy and B language. Hence this will lead to tedious repeated modelling each time a change has to be made between the design and analysis phase. This in turn is one of the motivations for this research, which is to reduce the tedious repeated modelling work done when two modelling language is involved.

As shown in 2.3 above, SD2PN consists of a very strong foundation to transform Sequence Diagram to Petri Net. However, this is only a one way process. Due to this limitation, this serves as another motivation for this research, which is to find a way to transform Petri Net back to Sequence Diagram. With SD2PN and the algorithm to transform Petri Net to Sequence Diagram, developers will be able to easily transform models between Sequence Diagram and Petri Nets.

The topic of bridging the gap between model design (UML) and model analysis (Petri Nets) will be further discussed in the following chapter.

CHAPTER 3

METHODOLOGY

This chapter discusses the methodology used in this research. The related methodologies used in this research includes the concept of Multi Paradigm Modelling and how they could be used to solve the problem statement presented in previous part of this thesis. The concept of Labelled Event Structures will also be presented in this chapter.

3.1 ROLE OF MODELLING IN SOFTWARE/SYSTEM DESIGN

The software design phase is not only a process but also a modeling phase. The design model is equivalent to the blueprint of the project. It provides guidance for constructing each and every detail in a system. By using models in software or system design, level of abstract can be raised and different types of views of the system can be observed. According to Van Gigch [27], there are three areas involved in system design, which is reality, modeling and metamodeling. Reality represents the view of the system in real life. Modelling is an abstraction of the reality by converting it into a verbal, graphical or mathematical notation. Lastly, metamodeling represents an abstraction of modelling or the modelling of the modelling process.

Other than raising level of abstract, models also function with different levels of formalism. As proven by J. Klein, F. Fleurey and J.M. Jezequel [28], there are three levels of formalisms, which are natural language, semi-formal notation and formal notation. Natural language models are very expressive and flexible simply because they comprise of descriptions and annotations that are easy to read and write. But the lack of semantics in natural language causes a major problem, which is the models could be understood differently by each stakeholders including the software designer and the client. In contrast, models with semi-formal notation use notational semantics to represent the structure and behavior of the system. One of the examples for semi-formal notation model is the Unified Modelling Language (UML). UML is an informal modeling language used to offer a standard and unified way to visualize the design of a system. Last but not least, models with formal notations are recognized for their precise semantics with underlying mathematical foundation. Examples of models with formal notations are Petri Nets [10], Alloy [29] and Z notation [30]. This type of model with formal notations is usually used to perform model analysis due to the fact that they are mathematical based.

The role of modelling is vital in software or system development. Modelling provides a way for developers to perform model design, model analysis and model synthesis. Other than performing model design via modelling, developers can also evaluate the structural rigidity and the behavioral properties of the system via model analysis modelling. The role of model synthesis is where two or more models with mutual elements could be put together in order to get a more complete interpretation of the system. The role of model design, model analysis and model synthesis will be presented in the following sections to show their importance in the role of modelling for software or system development.

3.1.1 MODEL DESIGN

Model design is the practice of representing a view of the system in the form of models. In this stage, developers usually use a semi-formal notational model because it provides a good balance between the ease-of-use and precision. Throughout the years, UML has become the go-to language in the model design phase.

As presented earlier in the thesis, there are many different types of model in UML that are divided into two main types, which is structural diagrams and behavioral diagrams. Examples of structural diagrams include class diagram and component diagram. A class diagram describes the organization of a system by presenting the system's classes, their attribute, operations and the relationship among objects while a component diagram shows how components are held together to form larger components and or software systems. On the other hand, examples of behavioral diagrams includes use case diagram and sequence diagram. A use case diagram represents the user's interaction with the system and shows the relationship between the user and the different use cases where the user is involved. Meanwhile, sequence diagram shows the interactions between the elements in the system.

The well-established set of semantics for each model type in UML allows developers to easily express their views of the system into models. Together with the fact that UML is an informal modelling language, UML models are easily understood between any stakeholders of the system without any prior knowledge on modelling or programming language.

3.1.2 MODEL ANALYSIS

Model analysis is an important stage as it can serves as a preliminary analysis of the system. By performing mathematical analysis on the models of the system, vital feedback could be obtained on whether there are any major structural design flaws or unwanted behavior in the system. This will allow the developer to correct the design flaws before the system is built, which in turn reduce the time taken and costs from having to re-build the system if any flaws were to be detected during implementation phase.

Due to the mathematical needs in the analysis process, modelling languages with formal semantics such as Alloy and Petri Nets are needed in this stage. Alloy is a declarative specification language for expressing complex structural constraints and behavior in a software system [29]. It is very suitable to perform structural analysis of a system. Petri Nets is a state-based modelling language that has a strong mathematical foundation. It is capable of modeling conflicts and concurrencies while performing different types of performance analysis. Other examples of modelling language with formal semantics include B language and Z notation. The modelling language mentioned above all has a strong mathematical foundation which makes them suitable for accurate computational analysis.

With model analysis, critical errors can be avoided in the system development process. Analysis such as liveness analysis, deadlock detection and boundedness analysis could be carried out to make sure that the system is free of unwanted behavior. According to a research done by Wieland et al, model analysis is proven to be vital in the computation of dependencies between states and risk analysis [31].

3.1.3 MODEL SYNTHESIS

Model synthesis is also known as model composition. It is the process of allowing two or more models to be put together based on a set of common elements. This is important because modern complex system needs to be broken down in the design phase, so that each module of the system is designed separately and independently of each other to reduce the overall complexity of the model. Models could also be built based on a particular perspective such as security or quality-of-service (QoS). By performing model synthesis amongst different modules or integrating the several perspectives of a system, an integrated view of the system can be produced. This in turn highlights the dependencies between the modules and viewpoints.

Model synthesis can also be used in a more enterprise systems in the form of a plug-in. There is also a concept called *refinement* in model synthesis where by a set of behaviors could be plugged into an existing model without having to redesign the whole model. As an example, while designing a secure system, a system designer can plug in different security protocols into the system design to discover the best fit for needs of the system without needing to create various models.

Notion of model synthesis is well-established in some of the modelling language such as Petri Net as shown in researches done by Yakovlev et al [32] and Agerwala and Choed-Amphai [33] where different techniques and algorithms are used for different types of synthesis. As an example, the refinement of a specific state in the Petri Net calls for a top-down [34] synthesis method using a place refinement or transition refinement algorithm [35].

3.2 HETEROGENEITY BETWEEN MODELLING LANGUAGE IN MODEL DESIGN, ANALYSIS AND SYNTHESIS

As shown in the 3.1, the role of modelling is important in system development in the design, analysis and synthesis phase. However, each phase have different requirements which leads to the use of different modelling language for each phase. As discussed in 2.4.3, this result in a condition called heterogeneity, where two modelling language UML (model design) and Petri Nets (model analysis) could not communicate with each other. In model design phase, modelling language used is in semi-formal notation, while in model analysis phase, modelling language is more mathematical-based as analysis carried out are mathematical-based. Due to the heterogeneity between the modelling languages, there is a lack of interoperability between the tools of the modelling languages. This presents as a serious challenge to system developers as shown in [36] and [37] to provide a platform that allows interoperability between different models with different levels of formalisms. One way to tackle this problem is via Multi Paradigm Modelling which will be explained in the following section.

3.2.1 MULTI PARADIGM MODELLING

Multi Paradigm Modelling is a platform that supports interoperability between heterogeneous models [38]. Vangheluwe et al applied Multi Paradigm Modelling in modelling and simulation [39] and describes Multi Paradigm Modelling as a field that focus on three directions of research, which is multi-formalism modelling, model abstraction and metamodeling [ ].

Multi-formalism modelling offers an interoperability platform for models with various levels of formalisms based on the foundation of model transformation. Model transformation is the practice of translating one model into another using a set of predetermined rules. Model transformation plays an important role in Model Driven Development [41]. It is intended to generate low-level models from higher level models, synchronize models with different levels of formalisms and reverse engineer higher level models to lower level models. The most common way to express a model transformation is by using QVT relational language [42] which is a standard for model transformation defined by Object Management Group (OMG). The main features that is common to all model transformations as shown by Czarnecki and Helsen [43] includes specification, such as the pre condition and post conditions for a model transformation, the set of transformation rules, the directionality of the transformations and also the source and target relationship. In MDD model transformation, the source metamodel and the target metamodel are required whereby each source and target model should conform to respective metamodels.

Model abstraction is the practice of removing a certain low-level detail from the model while preserving the construct and general behavior of the system. Model abstraction is similar to multi-formalism modelling as it also uses model transformation. The major difference between the two model transformations is that in model abstraction, the source and destination models are of the level of formalism. Model abstraction is usually used in the removing of different complicated low-level behaviors in the system according to the requirement of a specific perspective. As an example, a complete model of the system with low-level behavior might be too complicated for distribution to other stakeholders. Using model abstraction, the model of the system could be simplified up to a certain level without losing the structural properties and important behaviors of the model.

Metamodelling is the modelling of models. Metamodel is a model that defines other models. As an example, suppose a modelling language $\mathcal{L}$ has a metamodel $\mathbb{M}_{\mathcal{L}}$. As such, $\mathbb{M}_{\mathcal{L}}$ is a model that describes the constructs of the language $\mathcal{L}$ and every model that is written with the language $\mathcal{L}$ must be an instance of the metamodel $\mathbb{M}_{\mathcal{L}}$. The metamodel of a modelling language can be considered as a specification for the language which also can be used for documentation purpose or a foundation for model analysis. With metamodeling, new languages can be born just by modifying or tweaking parts of the existing metamodels. This in turn will allow customization of the modelling language to serve for a specific purpose.

3.2.2 USING MULTI PARADIGM MODELLING TO BRIDGE THE GAP BETWEEN MODEL DESIGN, ANALYSIS AND SYNTHESIS

As shown in Chapter 1, the main concern of this research is to bridge the gap between Sequence Diagram, Petri Nets and SD2PN. SD2PN provides a set of transformation rules to transform Sequence Diagram to Petri Nets which shows that it implements the concept of Multi-Formalism Modelling in the transformation phase. Similarly, my research will be based on the concept of Multi-Formalism Modelling, which means that a set of rules will be created to transform Petri Nets to Sequence Diagram.

UML Sequence Diagram is chosen as the language for model design because of it being able to model complex behavioral properties and interaction. Petri Nets is chosen as the language of model analysis because it is able to model dynamic behavioral models and its extensive capacity in model analysis.

With Multi Paradigm Modelling as a platform, Petri Nets models from model analysis could be transformed into Sequence Diagram models using a set of transformation rules. This in turn will create model interoperability between Petri Net models and Sequence Diagram models. The model interoperability between Petri Net models and Sequence Diagram models will allow system developer to use transformed Petri Net models for model analysis, make altercations to it, then perform model transformation and transform it back to Sequence Diagram model instead of manually updating the Sequence Diagram model. Figure 18 illustrates how does this proposed method improves the current methodology in creating a system.

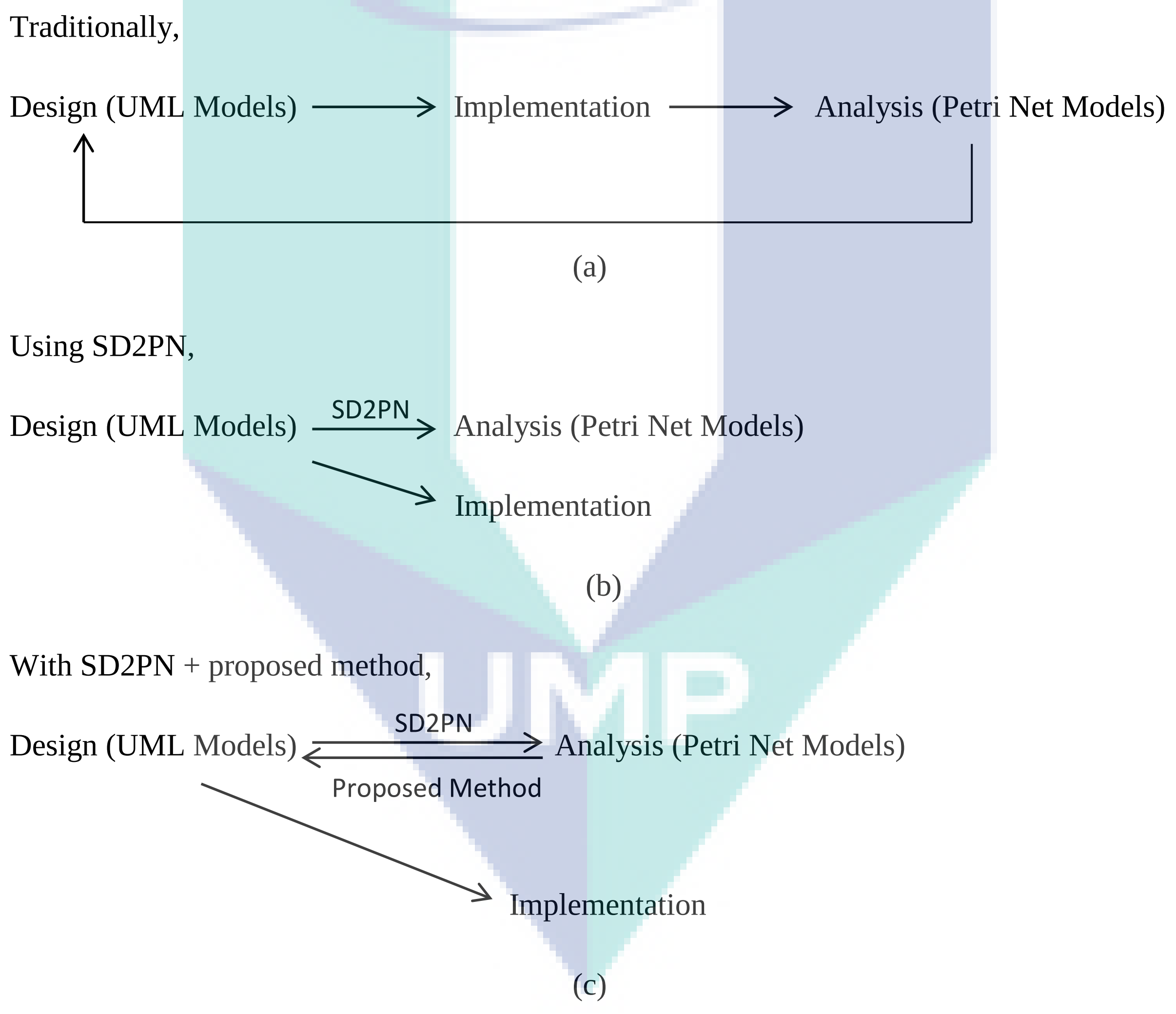


Figure 18: Contribution of the Proposed Method

Figure 18 (a) illustrates the traditional methodology in creating a system. Typically in the design phase, UML are chosen to be the modelling language. Details and specifications of the system are modelled in UML during the design phase. The implementation phase is also known as the coding phase, where codes are generated based on the UML models provided from the earlier design phase. After generating the coding and implementation phase, the analysis phase will be carried out to check for any faults in the system. Due to the mathematical nature of the analysis, this phase usually involves formal modelling language which is mathematical based, such as Petri Nets and Alloy. If any faults or errors are to be found in the analysis phase, developers are required to go back to the design phase and back track the problem which is time and cost consuming.

With the aid of SD2PN in Figure 18 (b), developers are able to transform UML Sequence Diagram models from design phase into Petri Net models. With the Petri Net models generated, developers can now use it to perform analysis. Indirectly, the analysis phased is carried forward with the use of SD2PN. Analysis can be performed on the Petri Net models generated from SD2PN which in turn will reduce errors in the implementation phase. However, developers still need to manually update the Sequence Diagram models as SD2PN only provide one way mapping, which is from Sequence Diagram models to Petri Net models. Upon performing analysis on the Petri Net models, developers still need to update the Sequence Diagram, which might be a tedious and repetitive process.

Figure 18 (c) highlights how the proposed method when combined with SD2PN will benefit the system developers. With the proposed method of transforming Petri Net models to Sequence Diagram models, developers can now use the transformation rules to transform Petri Net models back to Sequence Diagram. This eliminates the need to manually update the Sequence Diagram models after performing analysis on the Petri Net models.

The proposed method of transforming Petri Net models to Sequence Diagram models will be presented in the following chapter.

3.3 LABELLED EVENT STRUCTURES

In Multi Paradigm Modelling, it is very important to preserve the semantics of the models. To proof that semantics of the inputted Petri Nets are preserved in the semantics of the output

Sequence Diagram, we need a common semantic domain. In this research, labelled event structure is used as the common semantic domain since both Petri Nets and Sequence Diagram is able to be unfolded into Labelled Event Structure as proven by Küster-Filipe [44] and McMillan [45] respectively. Event Structures are models of computational process that allows a system to be modelled as a sequence of events, like a flow of events. Event Structure is able to model the behavior of a system through the relationship between the different events in the system. In Event Structure, there are three main types of relationship between events; they are causal relationship, conflicting relationship and concurrent relationship.

Definition 1: An Event Structure is a triple, $E = (Ev, \rightarrow^*, \#)$ where Ev is a set of events, $\rightarrow^*$ and $\#$ represents binary relation *causality* and *conflict* such that $\rightarrow^*, \# \subseteq Ev \times Ev$.

Causality is a partial order while conflict is symmetric, irreflexive and propagates over causality. If two events $e_1, e_2 \in Ev$ are neither in a causality or conflict, then they are concurrent, such that $e_1 co e_2$ *iff* $\neg (e_1 \rightarrow^* e_2 \vee e_2 \rightarrow^* e_1 \vee e_1 \# e_2)$.

Definition 2: An Event Structure $E = (Ev, \rightarrow^*, \#)$ is discrete *iff* for every e, the local configuration of e, $\downarrow e = \{e_n \mid e_n \rightarrow^* e\}$ is finite.

Immediate Causality refers to events such as $e_1, e_2 \in Ev$ that are causal and there are no other events occurring between them. If $e_1 \rightarrow^* e_2$ has an immediate causality relationship, then e_1 is the immediate predecessor of e_2 and e_2 is the immediate successor of e_1. Alternatively, this relation can also be written as $e_1 \rightarrow e_2$.

Definition 3: Let $E = (Ev, \rightarrow^*, \#)$ be a Discrete Event Structure and L an arbitrary set where $l: Ev \rightarrow L$ would be the labelling function that maps each event in E into an element in L.

From here on, Labelled Discrete Event Structures will be referred to as Labelled Event Structures or LES. The following part will introduce the process of unfolding Petri Nets to LES and also the process of translating Sequence Diagram to LES.

The following section briefly introduces and describes the process of unfolding Petri Nets into Labelled Event Structures and the process of translating UML Sequence Diagram into Labelled Event Structures.

3.3.1 UNFOLDING PETRI NETS INTO LABELLED EVENT STRUCTURES

McMillan [45] introduces a method that maps Petri Nets into LES based on a branching process of Petri Nets called unfolding. In this method, a net is created where the nodes in the net created are labelled by the elements of the original net. This net represents the firing sequence or a reachable marking of the original net. This net is also referred to as a Labelled Causal Net or a Labelled Occurrence Net and it can be interpreted as Labelled Event Structure.

Definition 4: Based on [45], suppose a Petri Net $N = (S, T, F)$, and a Labelled Occurrence Net (unfolding of N) consist of a Petri Net $N' = (S', T', F')$ with the labelling function L' which maps P' onto the set of P and T' onto the set of T while satisfying the following conditions:

- Well-foundedness: every subset of T must have a minimal element with respect to F^*.
- No forward conflicts: if $p \in P'$, $p \in t_i\bullet$ and $p \in t_2\bullet$ then t_1 and t_2 must be the same.
- No self-conflicts: if $t_1, t_2, t_3 \in T$, $t_1F'^*t_3$, $t_2F'^*t_3$ and $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, then $t_1 = t_2$
- No redundancy: if $t_1, t_2 \in T$, $L'(t_1) = L'(t_2)$ and $\bullet t_1 = \bullet t_2$, then $t_1 = t_2$

The construction of unfolding begins with the creation of a place for each places in the initial set and adding transitions for every set concurrent places corresponding to the input set of the original transition. From that transition, a place set corresponding to the output set of the original transition is generated and this process is done iteratively for the whole Petri Net.

For a more detailed explanation, an example of the process will be presented in section 4.3 of this thesis.

3.3.2 TRANSLATING UML SEQUENCE DIAGRAM INTO LABELLED EVENT STRUCTURES

The translation of UML Sequence Diagram into Labelled Event Structure is based on a research done by Küster-Filipe [44]. To represent Sequence Diagram as a Labelled Event Structure, a formalized notation for Sequence Diagram is required. The notations for a Sequence

Diagram followed by the definition of two local functions *scope* and *alt_occ* are in Definition 5, 6 and 7 correspondingly.

Definition 5: A Sequence Diagram can be represented as a tuple SD = (I, Loc, Loc_{ini}, Mes, E, $Path$, X_1), where

- I is a set of instance identifiers corresponding to the objects in the diagram
- Loc is the set of locations
- Loc_{ini} is the set of initial locations such that $Loc_{ini} \subseteq Loc$
- Mes is the set of message labels
- E is a set of edges where an edge (l_1, m, l_2) represents a message m sent from location l_1 to l_2
- { X_1} where $i \in I$ is a family of I-indexed sets of constraint symbols
- *Path* is a given set of well-formed path terms for the diagram used to capture the relative positions of the locations within a diagram

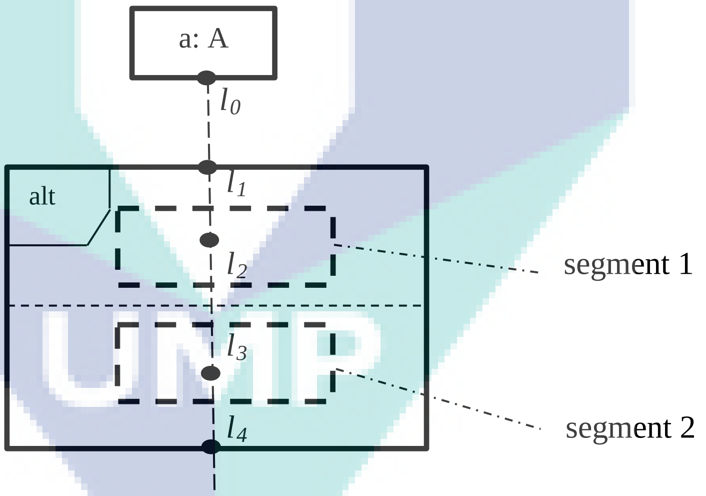


Figure 19: Example of events in a Sequence Diagram

Definition 6: *Scope* is a function given by *scope*: $Loc \rightarrow Path$. As show in Figure 19, *scope*(l_2) = alt(2)#1 and *scope*(l_3) = alt(2)#2. This can be further explained by showing that l_2 and l_3 are inside an alt fragment with two segments, however l_2 is in segment 1 and l_3 is in segment 2. Scope for l_1 and l_4 however indicates the start and end of a fragment and are shown as *scope*(l_1) = alt(2) and *scope*(l_4) = alt(2).alt(2).

Definition 7: *Alt_occ* is a local function given by *alt_occ*: *loc*(i) → ℕ that returns a possible number of alternative scenarios that can lead to a specific location. Based on Figure 19 above, *alt_occ*(l_4) = 2 since there are 2 possible scenarios that could lead to l_4 from the initial location of l_0 which are scenarios $S_1 = \{l_0, l_1, l_2, l_4\}$ and $S_2 = \{l_0, l_1, l_3, l_4\}$.

With the local function of *scope*, messages that are not causal and have a relationship of either *conflict* or *concurrent* can easily be identified. This information in turn would be important in the creation of the LES. Meanwhile with the local function of *alt_occ*, the number of alternative scenarios that leads to a specific location in the diagram can be acquired; this will in turn create the branches in the corresponding LES.

Based on the example in Figure 19, a fragment of LES that resembles to that particular Sequence Diagram can be created. Since l_4 has an *alt_occ* of 2, this means that it has two events associated to it, which is e_4 and e_5. The rest of the locations have an *alt_occ* of one, and is represented by e_1, e_2 and e_3 respectively. Hence, with the 5 events $Ev = \{e_1, e_2, e_3, e_4, e_5\}$ and $e_2 \# e_3$ as can be seen from the *scope*, a fragment of LES such that $\downarrow e_4 = \{e_1, e_2, e_4\}$, $\downarrow e_5 = \{e_1, e_3, e_5\}$ is the result, as shown in the following figure below.

$$\begin{tikzcd}
& e_1 \arrow[dl] \arrow[dr] & \\
e_2 \arrow[d] \arrow[rr, phantom, "\#"] & & e_3 \arrow[d] \\
e_4 & & e_5
\end{tikzcd}$$

Figure 20: Labelled Event Structure created corresponding to the Sequence Diagram in Figure 19

For a more detailed explanation, an example of the process will be presented in section 4.3 of this thesis.

CHAPTER 4

RESULT

This chapter will be focused on the proposed method in transforming Petri Net models to Sequence Diagram models. Mainly on how the algorithm works and what are the steps in transforming Petri Net to Sequence Diagram.

Petri Net is a formal and mathematical modeling language that is used for performing various types of analysis [10]. Petri Nets are mostly used to model control flow in a system and is capable of modeling conflicts and concurrencies. Sequence Diagram is an interaction based modelling language that describes the flow of events between objects in the system. It is a construct of message sequence chart [15]. These two modelling languages share some similar characteristic when carefully examined. The commonality between the languages makes them a candidate for implementing Multi Paradigm Modelling. The mathematical nature of Petri Nets provides a platform for analysis and manipulating the formal elements of the system while Sequence Diagram provides a user friendly, low-formalism platform for designing the system and communicating the system design with other stakeholders without prior knowledge to modelling and programming language. In this chapter, PN2SD (Petri Net to Sequence Diagram) will be introduced. PN2SD provides a framework for Petri Nets to be transformed into Sequence Diagram and serves as a basis for the Multi Paradigm Modelling.

4.1 PN2SD

Petri Net to Sequence Diagram (PN2SD) is a rule-based MDD model transformation that transforms any Petri Nets that conforms to the metamodel in Chapter 2 into Sequence Diagram. The process of the model transformation from Petri Nets to Sequence Diagram can be briefly explained as of below.

Step 1: Identify the Petri Net fragments in the inputted Petri Net.

Step 2: Transform each Petri Net fragments into a Sequence Diagram fragment based on a set of model transformation rules.

The steps involved will be further explained in details in the following sections.

4.1.1 IDENTIFYING THE PETRI NET FRAGMENTS

In SD2PN, *message* is considered to be a Sequence Diagram fragment. On the other hand, four types of InteractionOperatorKind are taken into consideration; they are *alternative*, *option*, *break* and *parallel*. Each of the four InteractionOperatorKind are able to change the flow of events in their own way, hence they are each designated as a fragment type. In total, SD2PN takes into the account of five types of fragments. To conform to SD2PN, this thesis will target similarly the five types of fragments in Petri Net; there are message, alternative, option, break and parallel.

To make the process of identifying the Petri Nets Fragments easier, some new elements are introduced in the notation of Petri Nets, which is the • symbol which indicates an arc. If the symbol appears before a labelling, it represents an incoming arc. If the symbol appears after a labelling, it represents an outgoing arc. For example, $S_1\bullet = 1$, this expression indicates the outgoing arc for the first state is one. Meaning there is only one outgoing arc attaching to S_1. Fragments in Petri Nets can be easily identified with the use of this simple notation.

4.1.1.1 PARALLEL FRAGMENT IN PETRI NET

Assuming there is (*n*) numbers of *S* (state) and *T* (transition) in the Petri Nets, for identifying parallel fragments in a Petri Net, the notation below can be used.

If $S_1\bullet = 1$ and $T_1\bullet > 1$, then it is the start of the parallel fragment.

If $T_n = 1$ and $\bullet T_n > 1$, then it is the end of the parallel fragment.

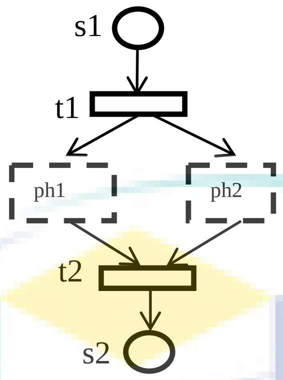


Figure 21: Parallel fragment in a Petri Net

Figure 21 shows a parallel fragment in a Petri Net. Based on the first expression of $S_1\bullet = 1$ and $T_1\bullet > 1$, the first state (S_1) has an outgoing arc and the first transition (T_1) has more than one outgoing arc, this depicts the start of a parallel fragment. When the transition T_1 has more than one outgoing arc, this indicates that the token from the earlier state is fired simultaneously based on the number of arcs or states connected to T_1. As for $T_n = 1$ and $\bullet T_n > 1$, this expression indicates that the (n)th number of T is one and the incoming arc to T_n is more than one, which in this case means when T_2 is one and the incoming arc into T_2 is more than one. This indicates that it is the end of a parallel fragment. In the parallel fragment in a Petri Net, two or more *placeholders* are present which represents the set of events in a parallel fragment. The *placeholders* are represented as *ph1* and *ph2* in Figure 21. The number of *placeholders* depends on the number of outgoing arc from T_1.

4.1.1.2 BREAK FRAGMENT IN PETRI NET

Assuming there is (n) numbers of S (state) and T (transition) in the Petri Nets, for identifying a break fragment in a Petri Net, the notation below can be used.

If $S_1\bullet = 2$ and $S_{2/3}\bullet = \varnothing$, then it is a break fragment.

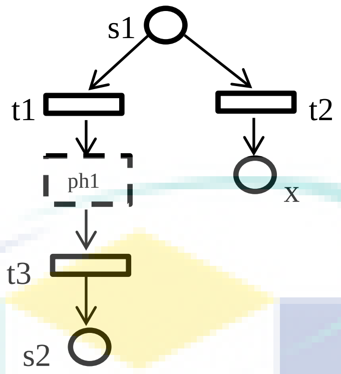


Figure 22: Break fragment in a Petri Net

Figure 22 shows a break fragment in a Petri Net. The expression of $S_1 \bullet = 2$ and $S_{2/3} \bullet = \varnothing$ indicates that if S_1 has two outgoing arc and one of the next two following states S_2 or S_3 do not consist of an outgoing arc, then it is a break fragment. In a break fragment, a state and x symbol is used to show the termination of the system which is also known as a terminal node. In a break fragment, only 1 *placeholder* exists since the function of a break fragment is similar to an 'if… else…' function in programming terms. Since the second choice is a termination of the system, no other *placeholder* is present.

4.1.1.3 ALTERNATIVE FRAGMENT IN PETRI NET

Assuming there is (*n*) numbers of *S* (state) and *T* (transition) in the Petri Nets, for identifying an alternative fragment in a Petri Net, the notation below can be used.

If $S_1 \bullet \geq 2$, then it is the start of an alternative fragment.

If $\bullet S_n \geq 2$, then it is the end of an alternative fragment.

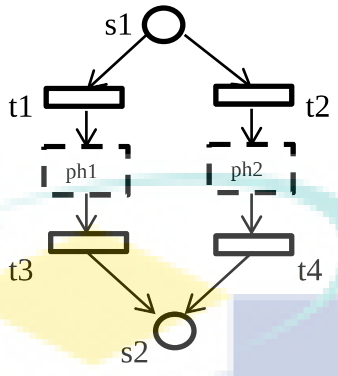


Figure 23: Alternative fragment in Petri Net

Figure 23 illustrates an alternative fragment in Petri Net. The notation of $S_1\bullet \geq 2$ indicates that if the S_1 has two or more outgoing arc, then it is the start of an alternative fragment. While the notation of $\bullet S_n \geq 2$ means if the incoming arc of the n(th) state has an incoming arc of two or more, then it is the end of an alternative fragment. In the example in Figure 23, $\bullet S_2 = 2$, hence it depicts the end of an alternative fragment. In an alternative fragment for Petri Nets, there can be two or more *placeholders* depending on the outgoing arc from S_1. If there are three outgoing arcs from S_1, then there will be three *placeholders* in the alternative fragment.

4.1.1.4 OPTION FRAGMENT IN PETRI NET

Assuming there is (*n*) numbers of *S* (state) and *T* (transition) in the Petri Nets, for identifying an option fragment in a Petri Net, the notation below can be used.

If $S_1\bullet \geq 2$, then it is the start of an option fragment.

If $\bullet S_n \geq 2$, then it is the end of an option fragment

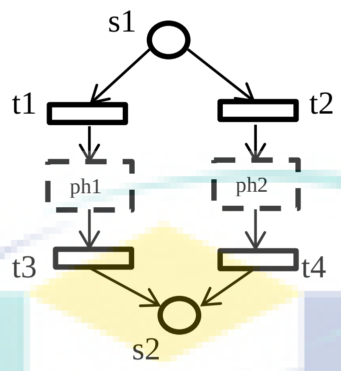


Figure 24: Option fragment in Petri Net

Figure 24 shows an option fragment in Petri Net. From the notation and figure above, the option fragment is similar to an alternative fragment based on their construct. Similar to alternative fragment, the expression of $S_1 \bullet \geq 2$ means that if the S_1 has two or more outgoing arc. This represents the start of an option fragment. On the other hand, the expression of $\bullet S_n \geq 2$ means if the incoming arc of the n(th) state has an incoming arc of two or more, then it is the end of an option fragment. In the example in Figure 24, $\bullet S_2 = 2$, hence it depicts the end of an option fragment. In an option fragment for Petri Net, there can be two or more *placeholders* depending on the outgoing arc from S_1.

4.1.1.5 MESSAGE FRAGMENT IN PETRI NET

A *message* represents the flow of information in the system between two objects. The notation below can be used in identifying a *message* fragment in Petri Net.

If $S_1 \bullet = 1$, $T_1 \bullet = 1$, $\bullet S_2 = 1$, then it is a *message* fragment.

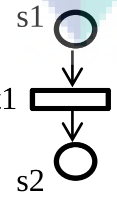


Figure 25: Message fragment in Petri Net

Figure 25 illustrates a message fragment in Petri Net. The expression of S_1• = 1, T_1• =1, •S_2 = 1 indicates that the outgoing arc for S_1 is only one, the outgoing arc from T_1 is one and the incoming arc to S_2 is one. When these three conditions meet, a *message* fragment is detected in Petri Net. If inside of a parallel, option or alternative fragment, the notation appears generally as S_n• = 1, T_n• =1, •S_{n+1} = 1.

The *sub_fragment* in break, parallel, alternative and option fragment can either be substituted with a *message* or another break, parallel, alternative and option fragment. The concept of *placeholders* will be further explained in the following section.

4.1.2 TRANSFORMATION RULES TO TRANSFORM PETRI NET FRAGMENTS TO SEQUENCE DIAGRAM FRAGMENTS

In this step, the transformation rules to transform the five types of Petri Net fragments to Sequence Diagram fragments will be explained. Five transformation rules will be presented, one rule for each type of fragment as introduced above.

In SD2PN, the concept of *placeholders* and Petri Net blocks are introduced. The concept of *placeholder* will be applied similarly in PN2SD. Each *placeholder* in a Petri Net fragment will be represented as a *sub-fragment* in a Sequence Diagram fragment. Each Petri Net fragment detected will be either transform into a *CombinedFragment* or a message fragment for Sequence Diagram.

4.1.2.1 RULE 1: TRANSFORMING PARALLEL FRAGMENT

The transformation rule for PN2SD is slightly different compared to SD2PN. In PN2SD, the four types of InteractionOperatorKind fragments are transformed first before transforming the *message* fragment. By doing so, the *CombinedFragment* of the Sequence Diagram can be identified first before proceeding to transform *message* fragments. A parallel fragment in Petri Net indicates that two or more sets of event should occur concurrently without any pre-defined set of conditions. There should not be any causality or conflicting event between all the operands of the parallel fragment.

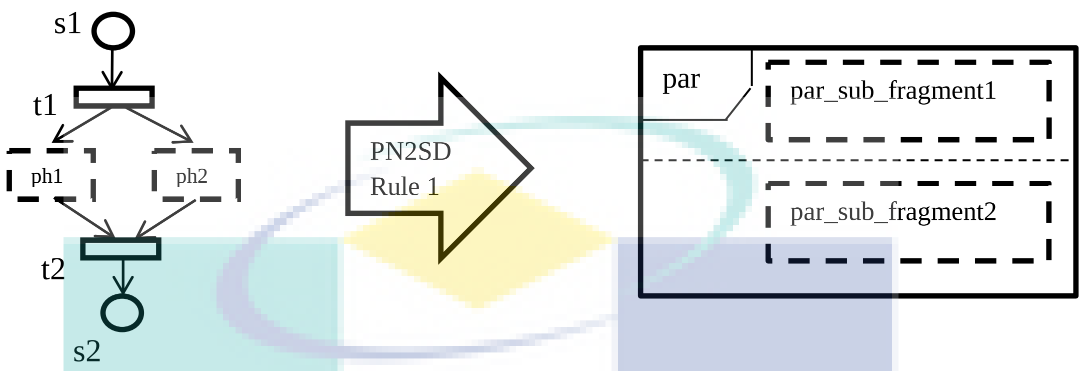


Figure 26: Applying PN2SD to a parallel fragment

Figure 26 illustrates how a parallel type of *CombinedFragment* for Sequence Diagram is created from a parallel fragment from Petri Net via model transformation rule for PN2SD. For each parallel fragment identified in Petri Net, a parallel type of *CombinedFragment* for Sequence Diagram is created. The parallel *CombinedFragment* consist of *sub_fragment* which represents the set of events in the *placeholder* (ph1 and ph2) from the parallel fragment in Petri Net. In other words, the *placeholders* from parallel fragment in Petri Nets are represented by *sub_fragment* in the *CombinedFragment* for Sequence Diagram. The *sub_fragment* inside a *CombinedFragment* can be substituted with either another *CombinedFragment* or a *message* fragment.

4.1.2.2 RULE 2: TRANSFORMING BREAK FRAGMENT

A *break* fragment in Petri Net consists of two choices, which is something similar to the 'if… else…' plus 'break' functions. In a *break* fragment, one of the *transition* leads to another set of Petri Net fragments while the other *transition* leads to the termination of the system which is also known as a terminal node. To illustrate the termination of the system, a *place* marked by X is shown. For each *break* fragment identified in Petri Net, a *break* type *CombinedFragment* in Sequence Diagram is created.

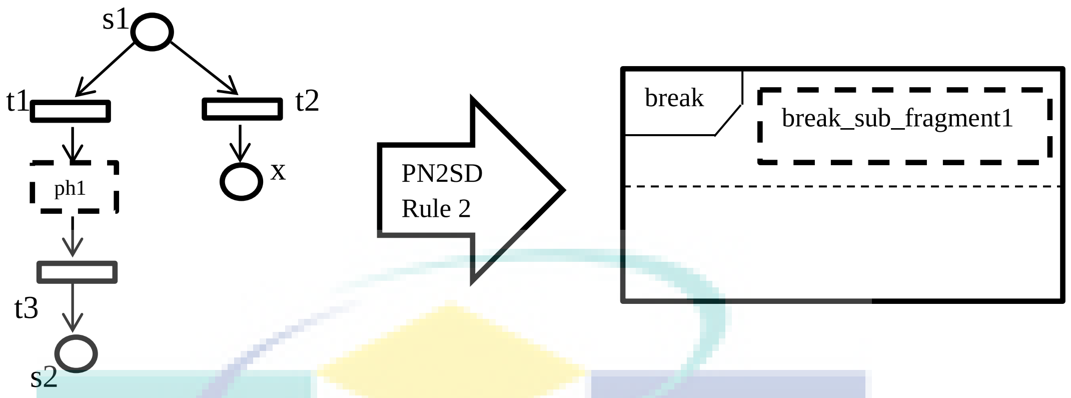


Figure 27: Applying PN2SD to a break fragment

Figure 27 shows how a *break* fragment from Petri Net is represented as a *break* type *CombinedFragment* in Sequence Diagram. The *placeholder* in the *break* fragment Petri Net is represented by the *sub_fragment* in the *break* type *CombinedFragment*. A *break* *CombinedFragment* consists of a guard (condition) such that when it is satisfied, the operation breaks. Notice that there are only one *sub_fragment* present in the *break* type *CombinedFragment*, this indicates that if *sub_fragment1* is not carried out, then the system is terminated.

4.1.2.3 RULE 3: TRANSFORMING ALTERNATIVE FRAGMENT

An *alternative* fragment in Petri Net also serves typically as an ‘if… else…’ condition in modelling interaction and behavior. For example if the event in *placeholder1* is carried out then the event in *placeholder2* will not be carried out. Alternatively, if the event in *placeholder2* is carried out, then the event in *placeholder1* will not be carried out. For each alternative fragment identified in Petri Net, an *alternative* type *CombinedFragment* in Sequence Diagram is created.

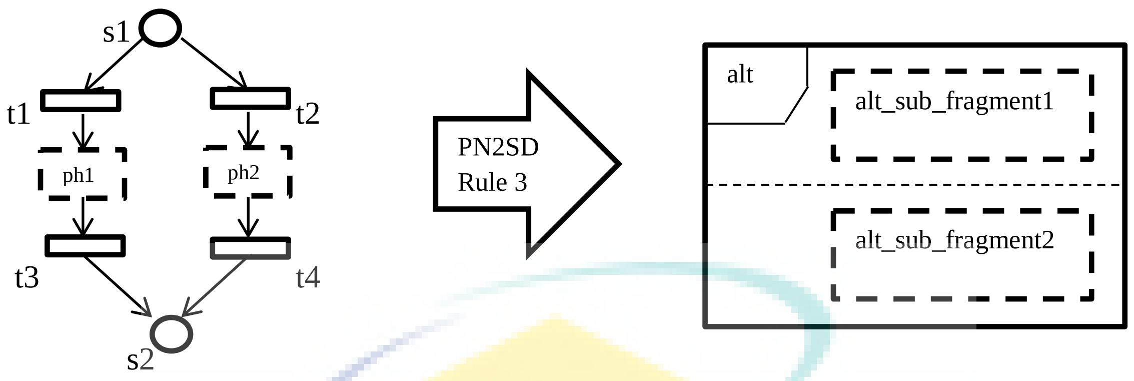


Figure 28: Applying PN2SD to an alternative fragment

Figure 28 above illustrates how an *alternative* fragment in Petri Net is represented as an *alternative* type *CombinedFragment* in Sequence Diagram. The two *placeholders* from the *alternative* fragment in Petri Net are represented by *sub_fragment1* and *sub_fragment2* respectively. Similar to the *alternative* fragment in Petri Net, in the *CombinedFragment* of *alternative* type in Sequence Diagram, when *sub_fragment1* is carried out, *sub_fragment2* will not be carried out.

4.1.2.4 RULE 4: TRANSFORMING OPTION FRAGMENT

The construct of an option *fragment* in Petri Net is similar to the *alternative* fragment in Petri Net. If the event in *placeholder1* is carried out then the event in *placeholder2* will not be carried out. Optionally, if the event in *placeholder2* is carried out, then the event in *placeholder1* will not be carried out. For each *option* fragment identified in Petri Net, an *option* type *CombinedFragment* in Sequence Diagram is created.

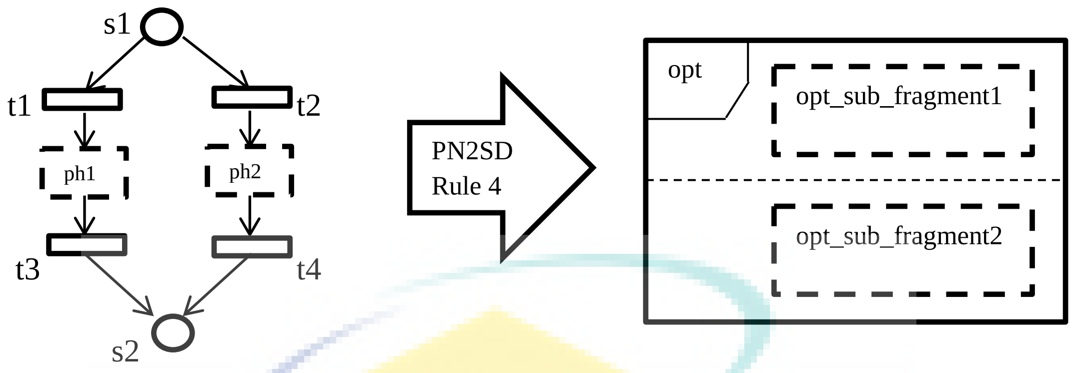


Figure 29: Applying PN2SD to an option fragment

Figure 29 shows how an *option* fragment in Petri Net is represented as an *option* type *CombinedFragment* in Sequence Diagram. The two *placeholders* from the *option* fragment in Petri Net are represented by *sub_fragment1* and *sub_fragment2* correspondingly. Similar to the *option* fragment in Petri Net, in the *CombinedFragment* of *option* type in Sequence Diagram, when *sub_fragment1* is carried out, *sub_fragment2* will not be carried out.

4.1.2.5 RULE 5: TRANSFORMING MESSAGE FRAGMENT

A *message* denotes the flow of information in the system between two objects. For each *message* fragments that exist in a Petri Net, an equivalent Sequence Diagram *message* is generated.

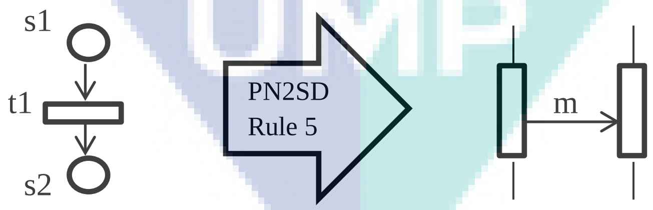


Figure 30: Applying PN2SD to a message fragment

Figure 30 above shows how a *message* fragment in Petri Net is represented in Sequence Diagram. The *state* (*s*) in Petri Net depicts the objects involved in the message which is represented by a *lifeline* in a Sequence Diagram. The *transition* (*t*) in Petri Net shows the movement of information from an object (*s1*) to another object (*s2*), which can be represented by an arrow showing the *message* flow in Sequence Diagram.

In the sub-topic 4.1, PN2SD which is a rule-based MDD model transformation that transforms any Petri Nets that conforms to the metamodel in Chapter 2 into Sequence Diagram is presented. Together with PN2SD, a full algorithm to transform Petri Nets to Sequence Diagram will be presented in the following part of this thesis.

4.2 ALGORITHM TO TRANSFORM PETRI NET TO SEQUENCE DIAGRAM

The flow chart below illustrates the algorithm to transform Petri Net to Sequence Diagram.

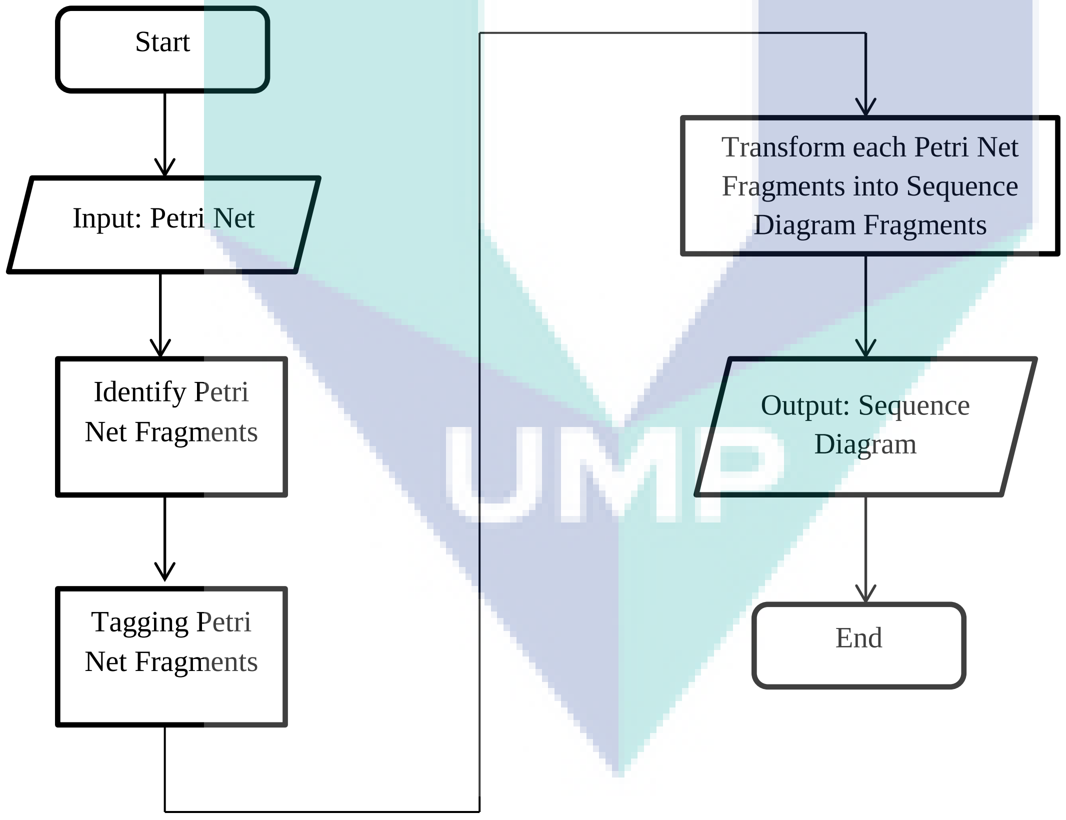

Start
Input: Petri Net
Identify Petri Net Fragments
Tagging Petri Net Fragments
Transform each Petri Net Fragments into Sequence Diagram Fragments
Output: Sequence Diagram
End

Figure 31: Algorithm to transform Petri Nets to Sequence Diagram

Figure 31 shows the algorithm to transform Petri Nets to Sequence Diagram in the form of a flow chart. A detailed explanation of the algorithm will be discussed in the following part of the thesis.

The algorithm begins by user inputting Free Choice Petri Net generated from SD2PN. Free Choice Petri Net is defined as a Petri Net where conflicts and concurrency may occur but not simultaneously. SD2PN generates only Free Choice Petri Net. Since the algorithm is designed to overcome SD2PN's limitation, hence, this algorithm will only accept Free Choice Petri Net as an input.

Upon inputting the Free Choice Petri Net, Petri Net Fragments are identified based on the method introduced in 4.1.1. In 4.1.1, a new concept is introduced whereby a • depicts an arc. If the symbol • appears before a labelling (either a state (*s*) or transition (*t*)), it represents an incoming arc. Where else if the symbol appears after the labelling, it represents an outgoing arc. As an example, $T_1 \bullet = 1$, this expression indicates the outgoing arc for the first transition is one. Meaning there is only one outgoing arc attached to T_1. With this newly proposed method, Petri Net Fragment such as option, alternative, parallel, break and message fragments can be easily identified. Details and examples of how to identify the Petri Net Fragments can be found in part 4.1.1 of the thesis.

After identifying the Petri Net Fragments, Petri Net Fragments are tagged to differentiate amongst each other. For example, when a *message* fragment is identified, it is tagged as m1, representing the first message. Subsequently, option, alternative, parallel and break fragments are tagged respectively as opt(n-th), alt(n-th), par(n-th) and bre(n-th), whereby the first option fragment is represented as opt1, second option fragment is represented by opt2 and so on.

The transformation process is next in line after the tagging process. In this process, each Petri Net Fragments that are identified will be transformed into a Sequence Diagram Fragment. As shown in 4.1.2, five transformation rules are introduced to transform the five different types of Petri Net Fragments detected. The concept of *placeholders* from SD2PN is also similarly applied in PN2SD. In PN2SD, each *placeholder* in a Petri Net fragment will be represented as a *sub-fragment* in a Sequence Diagram fragment. When compared to SD2PN's transformation process, the transformation process of PN2SD is slightly different. In SD2PN, Sequence

Diagram Fragments are transformed into Petri Net Fragments in one process and Petri Net Fragments are then morphed and substituted in another process. Meanwhile in PN2SD, transformations are done based on a top-to-bottom and left-to-right order. This eliminates the needs of another process to group the Sequence Diagram Fragments together.

Upon transforming the Petri Net Fragments into Sequence Diagram Fragments based on a top-to-bottom and left-to-right order, the output is a full Sequence Diagram. When a complete Sequence Diagram is obtained, the algorithm ends.

The description above briefly describes how the algorithm works. A more detailed application of the algorithm will be presented in the following chapter.

4.3 PN2SD PRESERVES SEMANTICS

In Multi Paradigm Modelling, preservation of semantics is important. It is important for the resulting Sequence Diagram to retain the same behavioral properties as the original input of Petri Net. In this part, the term correctness is referred as the preservation of semantics between the source model (Petri Net) and the destination model (Sequence Diagram).

To proof that the input of Petri Net Fragments and the output of Sequence Diagram Fragments consist of the same behavior, the semantics of both the Petri Net Fragments and Sequence Diagram Fragments are compared. Hence, a common semantic domain is needed in this case.

The chosen common semantics domain is the Labelled Event Structures (LES). Based on [44] and [45], we can observe that both Petri Net and Sequence Diagram can be represented as LES. Other than that, LES also offers a very similar method to modelling when compared to Petri Net and Sequence Diagram. All three languages emphasize on the behavior and the flow of the events.

$$\begin{tikzcd}
\text{Petri Nets} \arrow[r, Rightarrow, "\text{PN2SD}"] \arrow[d, "\alpha"'] & \text{Sequence Diagram} \arrow[d, "\beta"] \\
\text{LES} \arrow[r, leftrightarrow, "="] & \text{LES}
\end{tikzcd}$$

Figure 32: Using LES as a common semantics domain to prove correctness

Figure 32 illustrates how LES is used as a common semantics domain to prove correctness. The symbol α depicts a semantic map introduced by McMillan [45] while the symbol β represents another semantic map introduced by Kuster-Filipe [44] which is used to unfold Petri Nets. Petri Net fragments and Sequence Diagram fragments are able to be mapped into LES using the semantic maps introduced by respective authors above. Both LES is then compared and used as a proof that the PN2SD model transformation preserves the semantics of Petri Nets in the resulting Sequence Diagram.

The following section shows how Sequence Diagram fragments and Petri Nets are unfolded into LES using semantic map introduced by Kuster-Filipe and McMillan respectively.

4.3.1 TRANSFORMING PETRI NET FRAGMENTS AND SEQUENCE DIAGRAM FRAGMENTS FROM PN2SD TRANSFORMATION RULES INTO LES

In this section, Petri Net blocks and Sequence Diagram fragments from PN2SD transformation rules are transformed into LES. To prove that PN2SD preserves semantics, both the LES generated from the Petri Net and Sequence Diagram needs to be equal. These processes will be shown in the following section.

4.3.1.1 MAPPING PETRI NET FRAGMENTS FROM PN2SD TRANSFORMATION RULES INTO LES

The translation of Petri Net fragments into LES uses the concept of unfolding which was proposed by McMillian [45]. Based on PN2SD, only the five types of fragments introduced in section 4.1.1 will be taken into consideration when mapping Petri Net fragments into Labelled Event Structure.

Message

The *message* fragment in Petri Net can be unfolded in a very straight-forward matter where the two states in the Petri Net are represented in the form of places s_1 and s_2. The causal relationship between the places ensures that the LES produced is such as in the Figure 33 in Section 4.3.2.

Alternative

The Petri Net fragment that represents the *alternative* fragment starts with a state represented by the place s_1 creating e_1 in the LES. Though, the conflict represented by the two outgoing arcs from s_1 signifies two conflicting events in the LES. As the *placeholders* are added to the LES to match the Petri Net, the resulting LES is represented in Figure 33in Section 4.3.2.

Option

The unfolding of an *option* Petri Net fragment is similar to the *alternative* fragment presented above.

Break

The unfolding of a *break* Petri Net fragment is similar to the *alternative* fragment but with only one *placeholder*.

Parallel

For a *parallel* Petri Net fragments, it starts with a place s_1 and ends with a place s_2. They can be depicted as events e_1 and e_2 correspondingly with e_1 forking out into the *placeholders* and merging at e_2. This results in the LES representation shown in Figure 33 in Section 4.3.2.

4.3.1.2 MAPPING SEQUENCE DIAGRAM FRAGMENTS FROM PN2SD TRANSFORMATION RULES INTO LES

The process of mapping Sequence Diagram fragments into LES uses the concept of semantics mapping which was proposed by Kuster-Filipe [44]. Based on PN2SD, only the five types of fragments introduced in section 4.1.1 will be taken into consideration when mapping Sequence Diagram fragments into Labelled Event Structure.

Message

In Sequence Diagrams, *message* fragment is defined by two events, which are e_1 the event that depicts the sending of the message, and e_2 that shows the receiving of the messages. Both events are causal, and both belong to the same *scope* (this is true for any case since messages are in horizontal and there can never be a scenario that the sending and receiving event of a message exist under different *scopes*). This will result in the LES as shown in Figure 33 in Section 4.3.2.

Alternative

The Sequence Diagram fragment of *alternative* has an initial location l_1 that represents the beginning of the fragment. Since an *alternative* Sequence Diagram fragment signifies two operands in the fragment, hence there are 2 scopes; alt(2)#1 and alt(2)#2. The location l_2 signifies the end of the *alternative* fragment. Since there are no other choices or concurrencies, so there is only one *alt_loc* for location l_1; hence the event e_1 is the starting point of the LES. However in location l_2, there are two possible *alt_loc* since the *alternative* fragment creates two different scenarios. As a result, the location l_2 creates two events which are e_2 and e_3. As e_2 and e_3 are conflicting events, hence there are no sets of execution traces that contain both events; subsequently the symbol # is placed between the events denoting conflicting behavior. Upon adding *placeholders* to represent *placeholders* in Sequence Diagrams, the LES created will be presented in Figure 33 in Section 4.3.2.

Option

As shown in Section 4.1.2.4, the *option* fragment is semantically equivalent to the *alternative* fragment, hence the transformation from an *option* Sequence Diagram Fragments to LES is similar to the transformation of an *alternative* Sequence Diagram fragments.

Break

A *break* Sequence Diagram fragment has a similar construct to the *alternative* Sequence Diagram fragment, but with only one *placeholder*. However it still consists of two locations which is l_1 and l_2 where l_1 has an *alt_loc* of 1 and l_2 has an *alt_loc* of 2, which in turns generates two conflicting events e_2 and e_3 which will be presented in Figure 33 in Section 4.3.2.

Parallel

A *parallel* Sequence Diagram fragment has an initial location of l_1. It indicates the beginning of the parallel fragment. There exists 2 scopes inside the fragment, which are par(2)#1 and par(2)#2 as shown in Section 3.3.2. These scopes represent the parallel events that occur inside the fragment. After these events are executed, a location l_2 signifies the end of the fragment. Since both the l_1 and l_2 has an *alt_loc* of 1, so there is only 1 event to represent each of these locations, e_1 and e_2 such that e_1 forks into the 2 scopes of events and merge into e_2. This will create an LES which will be shown in Figure 33 in Section 4.3.2.

4.3.2 PROVING THAT PN2SD PRESERVES SEMANTICS

Upon generating LES from both Sequence Diagram fragments and Petri Net fragments, both of them need to be compared and made sure they equal. This is proven using the following Lemmas.

Lemma 1: Every Petri Net fragments and its corresponding Sequence Diagram fragments created by PN2SD generate the same LES.

Proof: As established in the earlier part of this thesis, there are five types of Petri Net fragments; *message*, *alternative*, *option*, *break* and *parallel*. Since both of *alternative* and *option* fragments are semantically equivalent, they will be grouped as one fragment for the purpose of this proof.

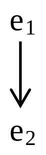
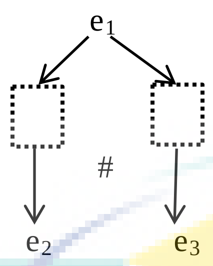
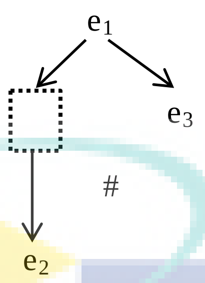
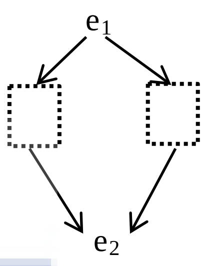
Message	Alternative / Option	Break	Par
			

Figure 33: LES obtained from Petri Net fragments and each corresponding Sequence Diagram fragments

The five types of Petri Net fragments are unfolded into LES based on the semantic mapping of α and the corresponding Sequence Diagram fragments are translated into LES with the semantic mapping of β. Upon unfolding and translating the Petri Net fragments and Sequence Diagram fragments respectively, a Labelled Event Structure for each type of fragment is created as showin in Figure 33 above. The semantic maps for unfolding a Petri Net fragment and translating a Sequence Diagram fragment was presented in Section 3.3 while the application of the semantic maps on the five types of fragments (*message*, *alternative*, *option*, *break* and *parallel*) were presented in Section 4.3. An example using a *parallel* type Petri Net fragment is presented below.

In a *parallel* Petri Net fragment, it starts with a place s_1 as an initial location and ends with a place s_2. They can be represented as events e_1 and e_2 respectively with e_1 branching out into the *placeholders* and merging at e_2.

In a *parallel* Sequence Diagram fragment, l_1 represents the initial location. The location depicts the beginning of the fragment. There are 2 scopes inside the fragment, they are par(2)#1 and par(2)#2 as shown in Section 3.3.2. These scopes signify the parallel events that take place in the fragment. Upon the execution of these events, the location l_2 shows the end of the fragment. Both l_1 and l_2 has an *alt_loc* of 1, hence there is only 1 event to represent each of these locations, which is e_1 and e_2 such that e_1 branches into the 2 scopes of events and merge into e_2. This in turn creates the LES as shown in Figure 33. Both of the Petri Net fragments and Sequence Diagram fragments can be represented exactly the same as the representation in Figure 33, hence this proves that the transformation preserves the behavior of the original Petri Net.

In the following chapter, an example of the application of the PN2SD algorithm will be presented.

CHAPTER 5

APPLICATION OF THE ALGORITHM

This chapter shows how the algorithm can be applied on a Free Choice Petri Net. Since the main concern of this thesis is to overcome the limitation of SD2PN which is it is only a one way process; upon using the algorithm, Petri Nets can be transformed back into Sequence Diagram. The algorithm serves as a guideline on how to transform Petri Nets into Sequence Diagram.

5.1 PN2SD OVERCOMES THE LIMITATION OF SD2PN

As proven by M. A. Ameedeen, SD2PN promotes model interoperability between Sequence Diagram and Petri Nets. A system designer is able to model a system in Sequence Diagram using UML tools, and then uses SD2PN to transform the Sequence Diagram models into Petri Nets. Upon transforming Sequence Diagrams into Petri Nets, complex analysis can be performed on the system using Petri Net tools. However, system designers are required to update the Sequence Diagram manually. PN2SD offers a MDD model transformation that transforms Free Choice Petri Nets into Sequence Diagram.

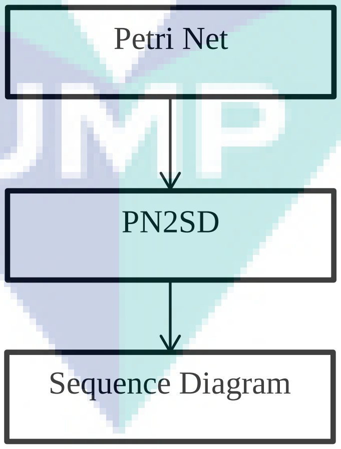


Figure 34: PN2SD transform Free Choice Petri Nets into Sequence Diagram

For system designer who used SD2PN to transform Sequence Diagram to Petri Nets for performing mathematical analysis, upon performing the analysis and making changes to the Petri Nets, PN2SD can be used to transform the Petri Nets back into Sequence Diagram.

5.2 EXAMPLE

Based on [46], SD2PN was implemented in a use case scenario which involves the behavior of a Personal Area Network (PAN). The PAN consists of a wireless router and a number of stations. With the aid of SD2PN, an integrated Petri Net is generated as shown in the figure below.

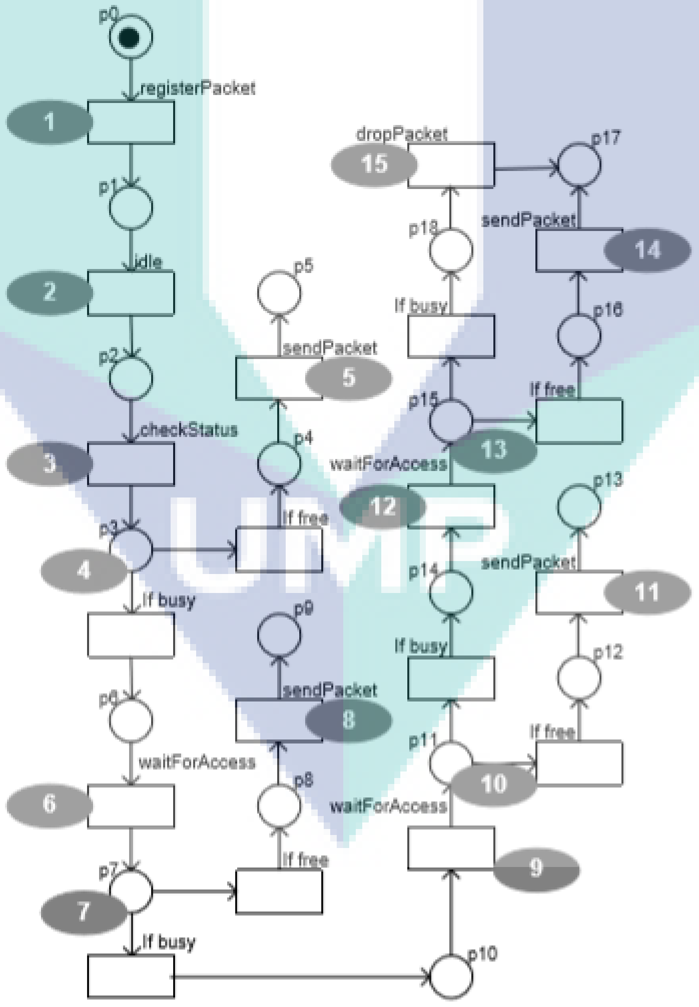


Figure 35: Petri Net for a station in PAN

Using the resulting Petri Net as an example, the algorithm to map Petri Nets to Sequence Diagram will be used to transform the Petri Net in Figure 35 into Sequence Diagram.

As shown in Figure 31 from Section 4.2, the algorithm to transform Petri Nets into Sequence Diagram begins by accepting Free Choice Petri Net as an input. In this case, SD2PN only generates Free Choice Petri Nets, since the author is using the Petri Net generated from SD2PN, the Petri Net is accepted as an input. The second step in the algorithm is to identify the types of fragments in the Petri Nets; each types of fragment are identified using the method presented in Section 4.1.1. Five types of fragment will be identified in this stage, they are *message, parallel, alternative, option* and *break* fragment. Next up, the fragments will be tagged. When a *message* fragment is identified, it is tagged as m1, representing the first message. Subsequently, option, alternative, parallel and break fragments are tagged respectively as opt(n-th), alt(n-th), par(n-th) and bre(n-th), whereby the first option fragment is represented as opt1, second option fragment is represented by opt2 and so on. Upon identifying the fragments, a rule-based MDD model transformation PN2SD is performed to transform the Petri Net fragments into Sequence Diagram fragments. The transformation process of PN2SD is slightly different compared to SD2P; in PN2SD, transformations are done based on a top-to-bottom and left-to-right order. This eliminates the needs of another process to group the Sequence Diagram Fragments together. After the transformation process, a Sequence Diagram is generated as shown in the figure below.

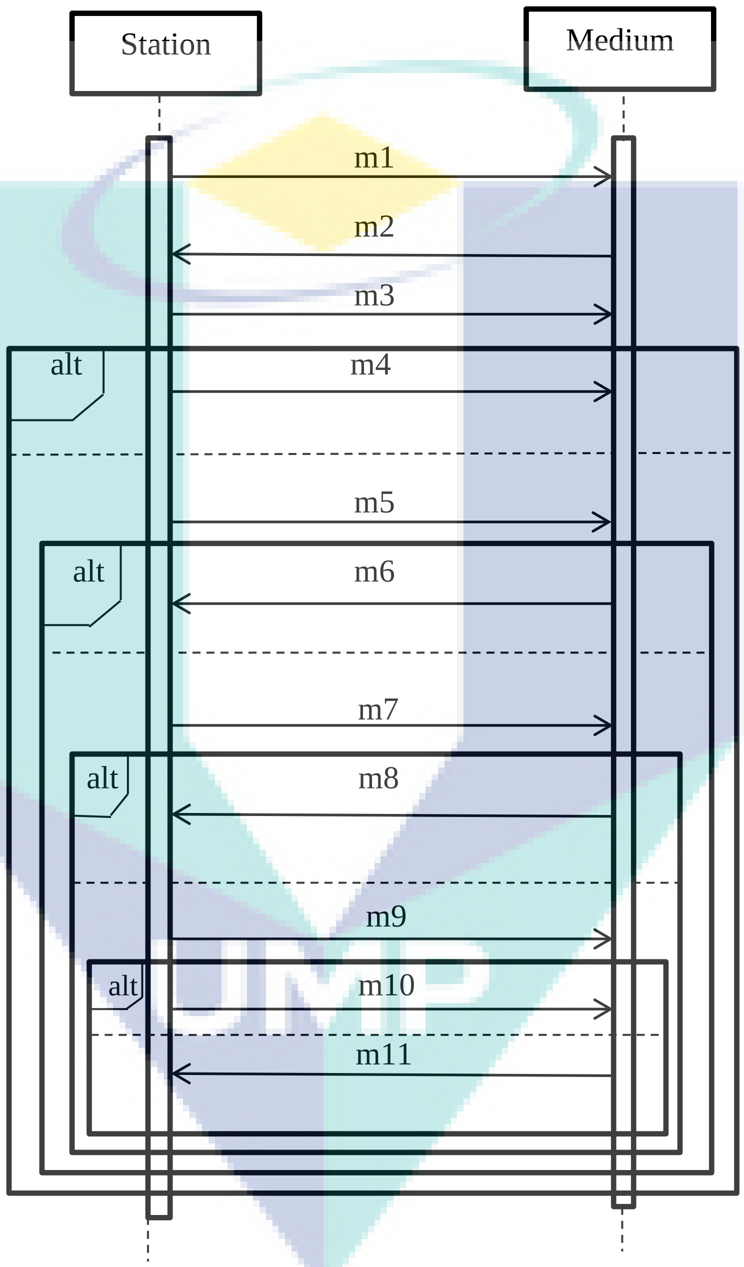


Figure 36: Sequence Diagram generated based on Figure 35

Comparison can be made based on the original Sequence Diagram used in [46]. Both Sequence Diagrams consist of the same number of *messages* and *alternative* operator fragments. The original Sequence Diagram used for SD2PN will be presented in Appendix A.

5.3 DISCUSSION

The example presented in this chapter shows that PN2SD is capable of transforming Petri Nets into Sequence Diagram. The algorithm serves as a guideline in transforming Petri Nets into Sequence Diagram. Referring to Section 4.3, it is also proven that PN2SD preserves semantics. By using the Labelled Event Structure as a common semantics domain, both Petri Nets and Sequence Diagram can be represented as LES. Both produce LES that is equal, which indicates that PN2SD can preserves semantics. A detailed explanation on how both Petri Nets and Sequence Diagrams are represented as LES can be obtained from Section 4.3.1.

Though, there is a limitation in PN2SD; which is the *message* fragment produced in Sequence Diagram are ambiguous. In a Sequence Diagram, there are *lifelines* that show the instance of the *objects* involved while sending the *message*. This means that users can clearly see the direction of the *message*, whether it is from *object* A to *object* B or vice versa. However, in a Petri Net, there are *states* which do not indicate which instances of the *objects* are being involved in the flow of event. Meaning that, a *message* fragment for Sequence Diagram generated via PN2SD can be either from *object* A to *object* B or vice versa, which is also known as ambiguous. In this case, assumptions need to be made such as each *message* is a continuity from the same object.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

This chapter presents the conclusion, contribution and limitation of the research work. Recommendations for future work are also available in this chapter.

6.1 CONCLUSION

PN2SD fills the knowledge gap between Sequence Diagram, Petri Nets and SD2PN. It is able to overcome the limitation of SD2PN, which is it only provides one way transformation from Sequence Diagram to Petri Net. PN2SD allow users to transform Petri Nets to Sequence Diagram. When used together with SD2PN, users are able to transform Sequence Diagram to Petri Nets to perform mathematical-based analysis. Upon performing analysis and modifying the Petri Nets, now users are able to use PN2SD and the algorithm to transform Petri Nets back to Sequence Diagram. This eliminates the need to manually update the Sequence Diagram each time a change is made on the Petri Net during the analysis.

Other than that, PN2SD provides a rule-based MDD model transformation from Petri Nets to Sequence Diagram. This in turn solves the heterogeneity between Petri Net and Sequence Diagram.

In the proposed algorithm, a new method is introduced to identify Petri Nets fragment. This is important as Petri Nets are not like Sequence Diagram, where each interaction operator is labelled accordingly. In Petri Nets, there are no labels that indicate *alternative*, *parallel*, *option* and *break* fragment. Future researchers will be able to benefit from the newly proposed method in identifying fragments in Petri Nets.

6.2 CONTRIBUTION

The major contribution of this thesis is presenting a way to transform Petri Nets to Sequence Diagram. This addresses the limitation of SD2PN with a newly proposed algorithm and rule-based MDD model transformation – PN2SD. PN2SD is an MDD model transformation that serves as a basis for Multi Paradigm Modelling between the Petri Nets and UML Sequence Diagram.

When PN2SD is used together with SD2PN, the traditional method of creating a system (Design, Implementation, and Analysis) can be enhanced. With PN2SD and SD2PN, the analysis phase can be carried forward to just before the implementation phase. Upon designing the system in UML Sequence Diagram, users now are able to use SD2PN and transform Sequence Diagram to Petri Net to perform mathematical-based analysis. This in turns reduces mathematical errors such as deadlock and liveness. After performing analysis on Petri Nets and modifying it, users can now use PN2SD to transform Petri Nets back to Sequence Diagram. This eradicates the need to manually update the Sequence Diagram. In short, mathematical-based errors can be prevented during the implementation phase. Cost and time can be saved.

6.3 LIMITATION

As stated in Section 5.3, the main limitation of PN2SD is the ambiguous *message* produced. When generating *message* fragments from Petri Net, the *message* fragment generated in Sequence Diagram is ambiguous. Assumptions need to be made when figuring out the direction flow of the *message* in Sequence Diagram.

Other than that, the fact that PN2SD only consider five types of fragments which is *message*, *alternative*, *option*, *parallel* and *break* is also a limitation. There exists other behavior such as *repeat*, which were not taken into account when designing the model transformation rules for transforming Petri Nets to Sequence Diagram.

6.4 FUTURE WORK

This section shows the ideas of the author to further continue in the current research carried out in this thesis. Firstly, a method to determine the direction of the *message* fragment generated via PN2SD should be looked into. With the proposed method of identifying the direction of the *message* fragment generated, a more accurate result can be obtained.

Other than that, the author also recommends a tool that integrates both SD2PN and PN2SD to provide a fully automated way to seamlessly transform Sequence Diagram to Petri Nets and vice versa. This in turn will reduce the time needed to manually transform Petri Nets back into Sequence Diagram.