A MODEL DRIVEN APPROACH TO ANALYSIS AND SYNTHESIS OF SEQUENCE DIAGRAMS

by

MOHAMED ARIFF AMEEDEEN

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
The University of Birmingham
December 2011

Abstract

Software design is a vital phase in a software development life cycle as it creates a blueprint for the implementation of the software. It is crucial that software designs are error-free since any unresolved design-errors could lead to costly implementation errors. In an approach to minimize these costly errors, the software community adopted the concept of modelling from various other engineering disciplines. Modelling provides a platform to create and share abstract or conceptual representations of the software system – leading to the birth of various modelling languages, among them Unified Modelling Language (UML) and Petri Nets. While Petri Nets strong mathematical capability allows various formal analyses to be performed on the models, UMLs user-friendly nature presented a more appealing platform for system designers. Using Multi Paradigm Modelling, this thesis presents an approach where system designers may have the best of both worlds; SD2PN, a model transformation that maps UML Sequence Diagrams into Petri Nets allows system designers to perform modelling in UML while still using Petri Nets to perform the analysis. Multi Paradigm Modelling also provided a platform for a well-established theory in Petri Nets – synthesis to be adopted into Sequence Diagram as a method of putting-together different Sequence Diagrams based on a set of techniques and algorithms. Finally, the model transformation is enhanced to transform Sequence Diagrams with timing constraints into Timed Petri Nets to allow time-related analysis such as Quality of Service (QoS) and performance analysis.

Acknowledgement

'In the name of God, the Most Gracious, the Most Merciful'

Firstly, a very big thank you to my family - without their blessing and support, I would not have the courage to pursue my dreams. A hearty thank you to the Malaysian Government and Universiti Malaysia Pahang for providing me with financial support and the opportunity for me to further my studies.

I would like to thank my supervisor, BehzadBordbar for his constant support, encouragement and motivation as well as my thesis group members Mark Ryan and Antoni Diller for taking an interest and providing me with constructive criticisms. I would also like to thank RachidAnane for a fruitful research collaboration.

My gratitude to KyriakosAnastasakis for his help and guidance at the beginning of my research.A special mention to Nur Hana Samsudin for being a great friend and for her help in handing-in my thesis. Not to forget my fellow research students Seyyed, Mohammed, Vinoth, Damien, Mike, Zeyn, Peter, Vivek and many others for making my stay in Birmingham a lot more colourful.

From the bottom of my heart, Thank You...

TABLE OF CONTENTS

CHAPTER 3: MULTI PARADIGM MODELLING

CHAPTER 4: SD2PN – SEQUENCE DIAGRAMS TO PETRI NETS

CHAPTER 5: SEQUENCE DIAGRAM ANALYSIS VIA SD2PN

CHAPTER 6: SEQUENCE DIAGRAM SYNTHESIS VIA SD2PN

CHAPTER 7: SD2PN AND TIMELINESS PROPERTIES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Software engineering is a discipline that facilitates the development of computer software using various methods, tools and procedures. The primary goal of software engineering according to Sametinger [1] is the cost-effective production of high-quality software systems. In this respect, Sametinger outlines attributes such as reliability, robustness, user-friendliness, efficiency and maintainability as the yardstick by which to measure the quality of the software system. He further explains the phases involved in software engineering; management, specification, design, implementation, testing and maintenance.

The software design phase is vital in any software development life cycle as it translates the specifications given by various stakeholders of the system into a set of blueprints that the software implementation is based on. It is absolutely crucial for a software design to be free from errors as any unresolved errors in the software design leads to errors in implementation or *bugs* that could possibly waste countless precious resources such as time, man-hours spent programming, and development cost to fix.

A complex software design process is iterative and is done on different levels of abstraction. As such, it is very easy to accidentally overlook errors in the design. According to [2]most software errors occur during the design phase. These design errors, or *bugs* increases in number with the ever-expanding complexity of the software systems – not through carelessness of the designers, but due to the human brain having limited ability to manage complexity [3].

One of the efforts made by the software engineering community in order to reduce these design errors is to adopt the concept of modelling from other engineering disciplines. Modelling is the process of creating an abstract or conceptual representation of a design that can be presented in an easily understandable format based on specific modelling languages. A model is usually presented in either in graphical or mathematical format. In software engineering, models were never at the forefront of software design until recently, despite the complexity. With the emergence of modelling as a method of software design, the inevitable birth of various modelling languages occurred.

Languages such as Z [4],Alloy [5] and Petri Nets [6] offered a way to perform software design in the form of models, based on the various constructs offered in the respective languages. However, the main advantage of using these languages in software design is the ability to perform formal, mathematical analysis of the software design, reducing the possibility of costly design errors being carried into the implementation phase. The Zand Alloy languages are mainly textual (modelled using mathematical and logical statements) and are used to model static properties of a software system while Petri Nets is a graphical modelling language with a strong mathematical foundation that is capable of modelling diverse sets of behavioursincluding parallel, asynchronous, concurrent, hierarchical and stochastic as well as dynamic behaviours[6]. Although these formal modelling languages are

precise and allow for mathematical analysis of the software design such as *liveness* and *deadlock detection* as well as *reachability*, it requires a specific set of expertise to be able to create and to understand the models; and as such it is not well adopted by system designers' since it is not generally their forte.

The evolvement of modelling language brought Unified Modelling Language (UML) [7] to the forefront. UML is a family of languages, which is widely accepted as the *de facto* standard forsoftware system modelling. UML models can be used to specify the structure of a system, its behaviour and the constraints thatthe system must adhere to. Models in UML are instances of *metamodels*[1]. A metamodel includes system elements, theirrelationships and a set of rules to which every model must conform in order to be well defined. UML diagrams can be classified as either structural or behavioural diagrams. Each type of diagram is used to model a specific aspect of a software system. For example, a Class Diagram (a structural diagram) is used to model the different classes in the system, their attributes and operations, as well as how the classes relate to one another; where else a Sequence Diagram (a behavioural diagram) is used to model dynamic interactions in terms of messages passed between objects in the system. Although UML has conferred itself a preferred role in the software design community because of its graphical approach and user-friendly nature, the trade-off may has been the strong mathematical foundation in which formal analysis of the software design may be performed.

In an ideal environment, a software system designer could have the best of both worlds; the easy to use, widely accepted nature of UML coupled with a strong mathematical backbone such that in formal modelling languages, where various analyses may be performed on the designs. This opens the door for MultiParadigm Modelling. MultiParadigm Modelling

[1]Metamodels are themselves *models*, from whichmodels of systems are instantiated.

provides a platform for *model interoperability* where two or more models from different levels of formalisms or different languages could be used interchangeably. Multi Paradigm Modelling is based on the concept of Model Driven Development (MDD) [84]. Central to the concept of MDD is *model transformation*, where a number of *Transformation Rules* are used tospecify how various elements of one language are mapped into the elements of another language. The process of Model Transformation is carried out automatically via the software tools whichare commonly referred to as *Model Transformation Frameworks* [18‑20].

One example of Multi Paradigm Modelling is UML2Alloy [8], an MDD model transformation that transforms UML Class Diagrams into Alloy constraints using a specific set of transformation rules. This allows the system designer to design the models in the form of Class Diagrams using UMLs user-friendly interface, then transform the models into Alloy using the UML2Alloy tool to analyse the model, taking advantage of the robust analysis capabilities of Alloy. However, UML2Alloy only creates a platform for structural analysis to be performed on the software design since both Class Diagrams and Alloy are static languages that does not support dynamic behaviour modelling.

This became the motivation behind this thesis – to create a platform for analysis of dynamic models via model interoperability between two dynamic languages, namely UML Sequence Diagrams and Petri Nets. Sequence Diagrams are a UML version of Message Sequence Charts [13] and they are widely used in Software Engineering[14]. Sequence Diagrams can be used in modelling complex software systems as they provide a sequential listingof events and are also able to model parallelism and conflicts. Sequence Diagrams model the behaviour of the system through interaction or communication between the various objects of a software system and arranges them with reference to occurrence time. The quest for mapping Sequence Diagrams into a more formal language began with the choice of the

formal language. There are numerous formal languages such as Z, B, Alloy and Petri Nets. However choosing the right formal language depends not only on the standings or regards of the language in the modelling community, it mainly needs to reflect the analysis capability of the language. Languages such as Z, B and Alloy, even though are well-studied and are well-capable to perform numerous analysis on various types of models; are more suited to modelling static items with added logical constraints. As such, the choice of Petri Nets as the formal language is straight-forward since it is also a very well-studied language with a strong mathematical backbone that is instead capable of modelling dynamic behaviour and interactions through causal events as well as parallelism and conflicts.

There are three main components in a Petri Net; *places, transitions* and *arcs*. Places often depict the state of the system where else transitions often represent an action or a transaction that occurs. Arcs on the other hand, connect places and transitions in a directional manner. There are also strong theories and applications associated to Petri Nets such as analysis [9-16] and synthesis [17-26].

The model transformation to transform Sequence Diagrams into Petri Nets (SD2PN) is created by mapping the Sequence Diagram metamodel into the Petri Net metamodel using a set of transformation rules. As such, every Sequence Diagram that conforms to the Sequence Diagram metamodel can be transformed into a Petri Net. These Petri Nets can then be analysed using widely available Petri Net tools such as CPNTools[27], PIPE [28] and various others[29-35]. In this thesis, it is also proven that each Petri Net generated by SD2PN belongs to a special, well-studied sub-class of Petri Nets called Free Choice Petri Nets[36]. Free Choice Petri Nets are unique in Petri Net circles where conflicts and concurrencies may occur, but not simultaneously. This structural boundary between Free Choice Petri Nets and general Petri Nets allows for computations to be performed far more efficiently in Free Choice Petri

Nets where analysis such as *liveness* and *boundedness* could be performed in *polynomial* time [37] instead of the possible *exponential* complexity in analysis of general Petri Nets. Other analyses problems that are based on live and bounded Free Choice Petri Nets are also feasible in polynomial time based on the studies in [36, 38, 39].

Another common issue in MultiParadigm Modelling is concerning the accuracy of the model transformation. Model interoperability may not be achieved unless the entire semantics of the source model is preserved in the destination model – and that no emergent properties are incurred at the destination model. In SD2PN, the preservation of semantics that proves the correctness of the model transformation is presented via a common semantics domain. As there are strong theories that map both Sequence Diagrams [40] and Petri Nets [41] into Labelled Event Structures (LES)[2][42], it is chosen as the common semantic domain for proving the correctness of the SD2PN model transformation, and as such achieving model interoperability.

In creating the model interoperability between Sequence Diagrams and Petri Nets, it was observed that an opportunity for sharing the theories and applications between the two languages emerged. Synthesis, or the notion of putting together various elements based on specific sets of constraints has been receiving considerable attention [8, 43-47] in the UML community as it allows software design to be performed using different sets of models that could be put together at a later point. Since it has been established that software design is a complex process that could result in numerous costly bugs, it is a good approach to have separate models for each modules in the system to reduce the complexity. However, manual synthesis of these models is error-prone, tedious and redundant. As such, by adopting the

[2] LES are models of processes that are capable to model causality, conflict and concurrency properties between events.

well-studied notions of synthesis in Petri Nets, a similar notion of synthesis is introduced in this thesis – based on the Multi-Paradigm Modelling framework.

Taking inspiration from the prominent synthesis methods in Petri Nets – namely the *top-down* [16] and *bottom-up*[23, 48] synthesis methods, two Sequence Diagram synthesis methods of the same name are introduced. In the top-down method, an algorithm that replaces a single message in a Sequence Diagram with an entire Sequence Diagram is presented, called *message refinement*. In addition, two examples of the bottom-up method features an algorithm called *part decomposition* that replaces a *lifeline* or an object in Sequence Diagram with a different Sequence Diagram in order to explain the internal structure of said object and a domain-specific synthesis algorithm to simulate a *man-in-the-middle*type attack in any relevant systems.

Finally, an extension of SD2PN to include timeliness properties is presented. By enhancing the Sequence Diagram metamodel, the occurrences of events in Sequence Diagrams are allowed to have a timing constraint attached to them. This permits time-related analysis such as Quality-of-Service (QoS) analysis or performance analysis [49] to be performed on the design model before they implemented. The model transformation generates a flavour of Petri Nets called Timed Petri Nets [50] and can be analysed using numerous tools including CPNTools and PIPE mentioned earlier.

1.1 Problem Statement

The ever-increasing complexity of the software development process has presented software designers with a significant challenge. This complexity is due to various factors including the

variety of application domains, the variety of software platforms and the variety of the methods and the tools that support the software development process. This complexity is further compounded by the non-functional requirements for a software system to satisfy a set of specific properties, such as fault-tolerance and security. Many approaches and methods have been proposed as a way of addressing and reconciling these issues, as discussed in a survey presented in [51]. Of particular significance in the generation of software system is the need to facilitate a smooth transition from one domain of the software design process to the next. This dichotomy between the domains manifests itself in the multiplicity of formalisms, languages and software tools that are required for each phase. One of the main concerns of software developers is how to bridge the gap between the different underlying domains and allow for a seamless transition between them [52].An example of this challenge and one of the two main focuses of this thesis is the transition from a model design domain to a model analysis domain. This is very often critical, especially as it regularly involves incompatible domains of discourse.

In a normal scenario, the software designer models the software system in UML. This design is then translated into a formal language such as Alloy [53], Z [4] or Petri Nets [6]. The existence of two types of models, created using two different sets of tools and using two different languages is what is described as heterogeneity. Heterogeneous models cannot communicate with each other under normal circumstances and requires a completely different skill-set to design. For example, a software designer who is well-versed in UML may not necessarily be familiar with the formal language models in Petri Nets, and vice-versa. Heterogeneity also leads to tedious repeated modelling each time a change has to be made.

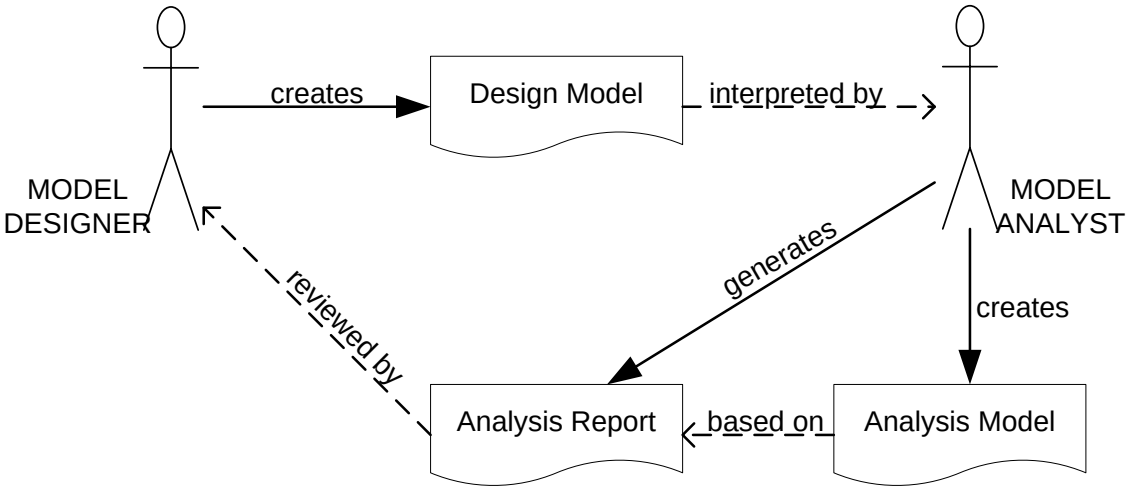

Figure 1: A typical scenario in the software design phase

Figure 1 above presents a likely example scenario where a model designer creates a design model using UML and passes it to the model analyst who interprets the design model and creates a corresponding analysis model. By performing a formal analysis on the model, the model analyst then generates an analysis report for the model designer to review. The model designer would then have to modify or create another design model. The whole process is repeated as long as there are errors found in the formal analysis.

A clear challenge presented in this scenario is to facilitate the transition between the design model and the analysis model automatically without having a model analyst to interpret the design model. This automated transition would not only eliminate the need for a model analyst, it will also create a more accurate formal representation for analysis since there would no longer be room for human misinterpretation of the design model. One possible way to achieve this automated transition between the design and analysis model is through MDD model transformation. This possible solution is explored in this thesis and makes up one of the two major contributions in this thesis.

Another problem that system designers are faced with is the influx of models. This could be on the basis of multi-viewpoint modelling or due to object-oriented design. To better manage essential complexity, developers of large software systems often create multiple models that describe the systems from a variety of perspectives. For example, developers may create multiple UML static and behavioural models, some describing core business functions, while others represent the system from a security[54]or quality-of-service (QoS) perspective. Some designers on the other hand, opt to model each component of the software separately to better identify critical sections of the software and keep the models manageable in size.

There are huge advantages to performing system design using multiple models, none more so than reducing the complexity of the designs where designers potentially deal with multiple instances of smaller, easier to design models instead of a single complex model for the entire system. Moreover for the multi-viewpoint models, each model could cater to the needs of a specific stakeholder as to gain better understanding of the software system through a specific point-of-view. On the other hand, for object-oriented design models, not only would it translate easily into object-oriented languages such as Java, but it also enables the system designers to identify errors and critical areas more easily and quickly. By using multiple models, error correction on the design level could be done easily – only the model that presents the error needs to be modified or maintained and all the other models would be unaffected.

Despite the advantages of performing system design using multiple models, during development, it may be necessary to incorporateall the different models describing different views.The process of incorporating different models together is called *synthesis*. A synthesized model provides an integrated view of the system, thus allowing developers to identify and analyze emergent properties that arise as a result of the integration. For example,

to support the evaluation of system's security features against an attack scenario, a model can be synthesized from two parts: one describing the attack scenario and the other the describing security features[54]. The composed model can be analyzed to determine how the security features withstand the attack. However, manual synthesis of non-trivial models can be tedious, error-prone, and redundant. The process is especially problematic when the models used for synthesis evolve; considerable effort is needed to manually synchronize such models. It has been established in [8, 43-47, 55-57] that automated support for model synthesis is needed for compositional development of models.

The challenge presented by this scenario is to automatically synthesize UML models using various algorithms to present an integrated view of the system. Unfortunately, the Unified Modelling Language (UML) [7] does not provide support for model synthesis in its language framework. However, the notion of synthesis is already well-established in formal languages such as Petri Nets[6, 17, 58, 59]. By utilising the aforementioned MDD model transformation as the basis, this thesis presents an approach to perform synthesis in UML. This signifies the second major contribution of this thesis. Meanwhile, other contributions are outlined in the following section.

1.2 Outline of Contributions

The contributions of this thesis are as follows:

- A new approach to bridging the gap between design and analysis of behavioural models via Model Driven Development (MDD). In particular:

- A model transformation called SD2PN that transforms UML Sequence Diagrams into Petri Nets (*refer page 63*).
- Two metamodels are identified; one for Sequence Diagrams and the other for Petri Nets (*refer pages 22 and 29*).
- A set of five transformation rules are defined to transforms fragments of Sequence Diagrams into blocks of Petri Nets (*refer page 65*).
- A concept of *placeholders* is defined in Petri Nets as a temporary node that mimics the connection capabilities of a *place*(*refer page 66*).
- Two local functions *morph* and *substitute* are defined to put together blocks of Petri Nets (*refer pages 77 and 79*).

- The SD2PN model transformation is mathematically proven to only generate Free Choice Petri Nets, a well-studied subclass of Petri Nets that is highly suited for analysis due to its low complexity (*refer page 82*).
- An approach for proving semantic equivalence between two domains is presented. More specifically:
  - A common semantic domain between Sequence Diagrams and Petri Nets is identified (*refer page 59*).
  - Using well established methods, both Sequence Diagrams and Petri Nets are mapped into the common semantic domain, which in this case is Labelled Event Structures (LES) (*refer page 84*).
  - The semantic preservation is established through comparison between the LES generated from the Sequence Diagrams and the LES created from the Petri Nets (*refer page 84*).

- An approach for analysis of Sequence Diagrams is presented using the SD2PN model transformation, based on the well-established mathematical analysis methods in Petri Nets (*refer page 89*).
- A tool for automated transformation of Sequence Diagrams into Petri Nets (SD2PN Transformer) is presented. In particular:
    - A method for parsing XMI data from well-known UML tools into Java objects is identified (*refer page 93*).
    - A Java code to perform the SD2PN model transformation is created (*refer page 166*).
- Synthesis methods for Sequence Diagrams are introduced based on Petri Net synthesis methods. More specifically:
    - A message refinement synthesis method that replaces a single message in a Sequence Diagram with a complete Sequence Diagram is presented (*refer page 118*).
    - A part decomposition synthesis method is defined as an example of a bottom-up synthesis method where the events attached to a certain lifeline are replaced with an entire Sequence Diagram (*refer page 123*).
    - A domain-specific, special case synthesis method used for introducing a man-in-the-middle type attack on a Sequence Diagram is presented (*refer page 127*).
- A notion of time is introduced into SD2PN, allowing various time related analysis such as performance analysis to be performed (*refer page 133*).
- A number of examples are presented in order to illustrate the feasibility of the studies presented in this thesis (*refer pages100, 118 and 149*).

1.3 List of Publications

The following publications are a result of the research presented in this thesis:

- Mohamed A. Ameedeen, Behzad Bordbar and Rachid Anane. Model interoperability via Model Driven Development. *Journal of Computer and System Sciences*. 2010.[60]
- Mohamed A. Ameedeen, Behzad Bordbar and Rachid Anane. A Model Driven Approach to Analysis of Timeliness Properties. *Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA 2009)*. 2009: Enschede, The Netherlands.[61]
- Mohamed Ariff Ameedeen and Behzad Bordbar. A Model Driven Approach to Represent Sequence Diagrams as Free Choice Petri Nets. *12th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*. 2008: München, Germany.[62]

1.4 Overview of this Thesis

The remainder of this thesis is organized as follows. In Chapter 2, preliminary and foundation information regarding UML, in particular Sequence Diagrams, followed by Petri Nets, Labelled Event Structures, Model Driven Development are presented.

In Chapter 3, a methodology for Multi Paradigm Modelling is presented, where else in Chapter 4, the application of Multi Paradigm Modelling in an MDD model transformation from Sequence Diagrams to Petri Nets (SD2PN) is presented. Chapter 4 also presents a

method for the automated transformation of Sequence Diagrams in the form of XMI to Petri Nets in the form of XML using a Java based tool. In Chapter 5, SD2PN is utilized to perform analysis of Sequence Diagrams using the mathematical analysis techniques in Petri Nets. Chapter 6 on the other hand brings the well-established notion of synthesis in Petri Nets into Sequence Diagrams. In particular, three synthesis algorithms are specifically introduced for Sequence Diagrams.

Chapter 7 introduces the concept of time into SD2PN, allowing time related analysis i.e. performance analysis to be performed using SD2PN. Finally, Chapter 8 summarizes the thesis and discusses the future work that can be done to extend this research.

CHAPTER 2
FOUNDATION

This chapter presents preliminary information or foundation for the languages and the technology used throughout this thesis including UML, Petri Nets and Model Driven Development.

2.1 Unified Modelling Language

Unified Modelling Language (UML)[7] is a family of languages, which is widely accepted as the *de facto* standard for software modelling. In the year 2003, UML was adjudged to be used in almost 70 percent of object-oriented software developments [63] and it is widely believed that this figure has ever since been steadily increasing. According to [64], the reason for this success is the six main advantages of UML as presented in Table 1.

Table 1: Main advantages of UML (referenced from [64])

Advantage	Description
Strongly defined	Every element used in UML has a strongly defined meaning provided in [7] and an explanation of how it could be used.
Concise	The notations used in UML are simple and straightforward, making the models clear, concise and simple.
Comprehensive	UML is built as a collection of languages that could describe various aspects of a system i.e. structure, behaviour, interactions, etc.
Scalable	UML is regarded to be strong enough to model large systems modelling projects. However, it is also adaptable to modelling smaller scaled systems without fuss.
Built on lesson-learned	UML is built based on the best practices in previous systems modelling methods. It is also constantly evolving for the better, taking note of other current system modelling methods as well.
Open standard	Since UML is built on an open standard with constant contributions from vendors and academics all over the world, it promotes interoperability and discourages a vendor monopoly.

One of the main goals of UML, as outlined in [63] is to provide an easy-to-use, expressive and visual languages that allows developers to design and share their models. In order to accomplish this goal, the UML standard [7] not only describes semantics of the

language, but also visual representation of the languages in the form of diagrams. UML diagrams are divided into two main classifications; structural diagrams and behavioural diagrams. Structural diagrams are commonly used to represent the architectural construct of the system by modelling the various structural and elemental properties. On the other hand, behavioural diagrams are used in modelling functional aspects of the system such as event flow and communication. Figure 2 presents the UML diagrams based on their classifications.

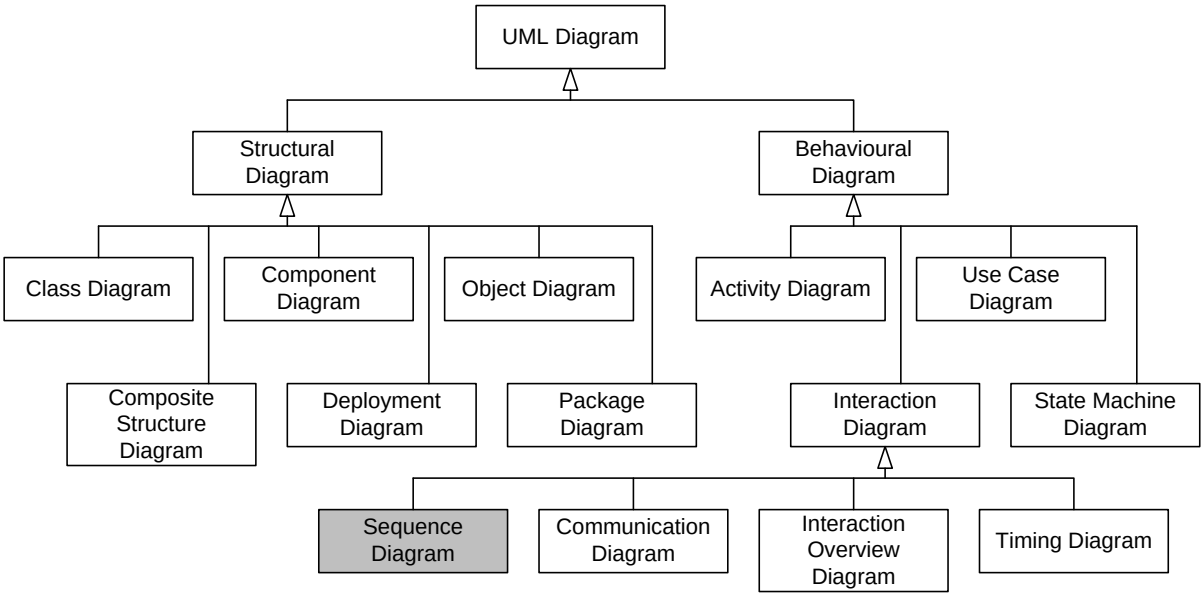

Figure 2: Classification of UML diagrams

Figure 2 classifies the six structural diagrams and seven behavioural diagrams, four of which is further classified as interaction diagrams. Table 2 lists and describes the different types of UML diagrams.

Table 2: Types of UML diagrams

Class Diagram (page 23 of [7])	A Class Diagram is a static diagram that depicts the structure of a system by representing the system in terms of

	classes and the relationship between them. A typical class in the Class Diagram is visually represented as a box with three sections. The top section holds the name of the class, the middle section lists the attributes of the class and the bottom section holds the methods that are associated to the class.
Component Diagram (page 143 of [7])	A Component Diagram is a diagram used to depict how various components in the system relates to one another. A component represents a modular part of the system and the relationship between the components, be it by connections or encapsulations are depicted in a Component Diagram.
Object Diagram (page 23 of [7])	An Object Diagram is used to represent a complete or partial view of a system modelled at a specific time. Objects Diagrams features object instances and attributes derived from Class Diagrams and the relationship between these instances. Object Diagrams are also used to provide examples or test-cases for Class Diagrams.
Composite Structure Diagram (page 161 of [7])	A Composite Structure Diagram is a type of static diagram that shows the internal structure of a classes and the collaboration between them. This diagram could be used to describe the *parts* (roles) of various instances, the *ports* (points) of connections between the classes, and *connectors* that are used to bind the entities together.
Deployment Diagram	A Deployment Diagram is used for modelling of the

(page 193 of [7])	physical deployment of *artifacts*. In UML, artifacts can be among others a model file, source file, a table or even a word document.
Package Diagram (page 23 of [7])	A Package Diagram presents the dependencies between packages in a model. It is commonly used to depict the architecture of a system using layers and the communication between them.
Activity Diagram (page 295 of [7])	Activity Diagram is used to present the workflow of activities in a system. It is capable of modelling iterations, choice as well as parallel behaviour. An Activity Diagram is often regarded as a form of flowchart.
Use Case Diagram (page 585 of [7])	A Use Case Diagram is a form of behavioural diagram that represents the overview of functionality in the system. A Use Case Diagram consists of *actors,* use cases and the dependencies between the use cases. A Use Case Diagram is often used to capture the requirements of a system.
State Machine Diagram (page 523 of [7])	The UML State Machine Diagram is a variation and extension of statecharts [65]. There are two types of state machines; behavioural state machines and protocol state machines. The behavioural state machines are used to specify behaviours of various model elements where else the protocol state machine is used to express usage protocols.
Sequence Diagram	Sequence Diagram is a type of interaction diagram that is

(page 457 of [7])	used to depict the communication between various object instances in the system. Sequence Diagrams are capable of modelling flow of events in a system as well as iterations, choice and parallelism.
Communication Diagram (page 457 of [7])	A Communication Diagram is a type of interaction diagram that is simplified from Collaboration Diagram in the previous versions of UML. It is commonly regarded as a combination between Class Diagrams, Sequence Diagrams and Use Case Diagrams as it is capable to model both static structures and dynamic behaviours.
Interaction Overview Diagram (page 457 of [7])	An Interaction Overview Diagram is used to model the control flow in a system (similar to Activity Diagram) using types of interaction diagrams (Sequence Diagrams, Communication Diagrams, Interaction Overview Diagrams and Timing Diagrams).
Timing Diagram (page 457 of [7])	Timing Diagram is a type of interaction diagram that focuses on timing properties. The horizontal axis of the Timing Diagram represents time, increasing from left to right where else the vertical axis represents the object instances.

Table 2 described the various types of UML diagrams. However, with reference to the highlighted element in Figure 2, the next section provides a more detailed view of a specific diagram type that will be used throughout this thesis; Sequence Diagrams.

2.1.1 Sequence Diagrams

Sequence Diagram is a type of UML Interaction Diagram adapted from its predecessor, Message Sequence Charts (MSC) [66, 67]. Sequence Diagrams are two-dimensional charts where the vertical axis represents the time and the horizontal axis represents interaction. Sequence Diagrams are commonly used to depict the flow of information in a system through communication between objects.

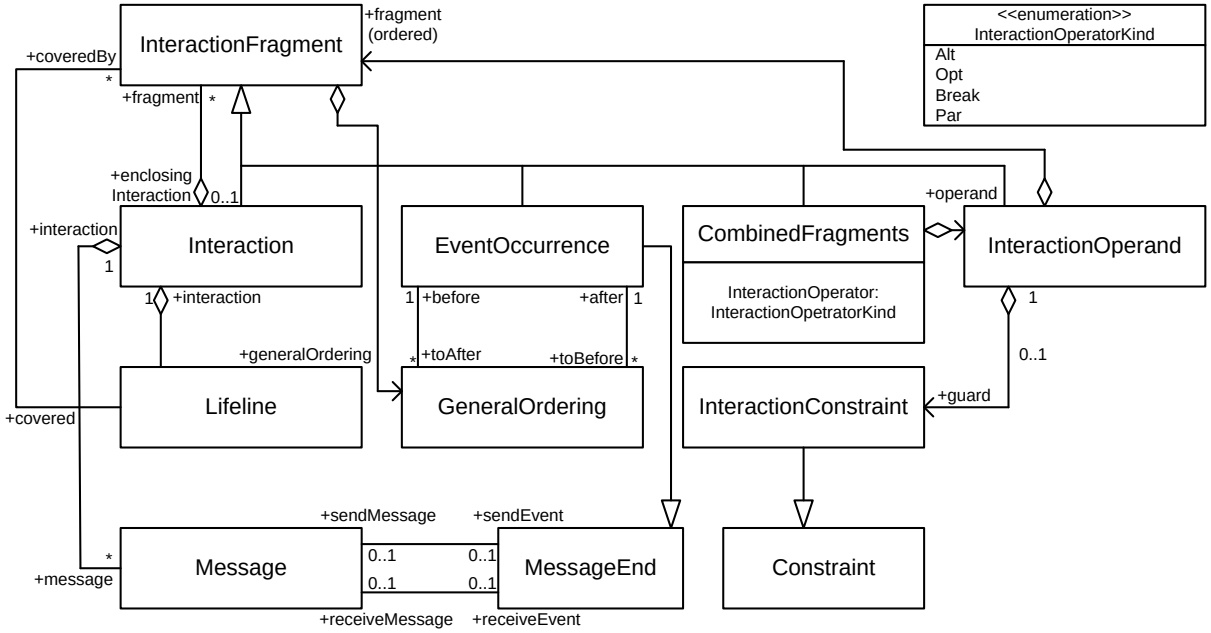

Figure 3: Sequence Diagram Metamodel

Figure 3 presents the metamodel for UML Sequence Diagrams featuring components of Sequence Diagrams as used in this thesis. The main components of a Sequence Diagram are *lifelines*, *messages* and *Combined Fragments* that are defined by *Interaction Operators*. Lifelines are horizontal lines that represent objects or instances of a class in the system where else messages are horizontal arrows that begins and ends at a lifeline. These messages represent the communication between the objects that are represented by the respective

lifelines. Messages are commonly used as a signal or a call for procedure or function in the system. Figure 4 presents an example of a Sequence Diagram with two lifelines and four messages. All four messages in Figure 4 depict communication from *Object A* to *Object B*.

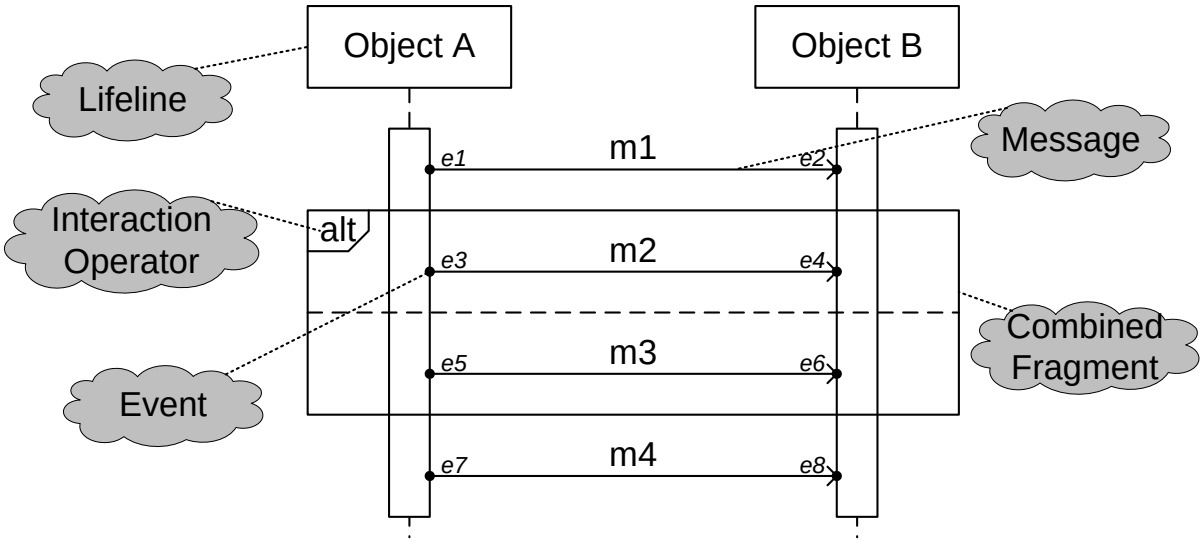

Figure 4: Example of a Sequence Diagram

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 number of operands it has. In Figure 4, a Combined Fragment with the Interaction Operator *alt* is presented with two operands (number of sections in the Combined Fragment). Interaction Operators are used as mechanisms to provide structure in the communication between lifelines. For example, the Interaction Operator *alt* presented in Figure 4 refers to *alternative* (conflicting) behaviour where only messages in one of the two operands pictured would be executed. As such, if the message *m2* is sent, then the message in the second operand, *m3* would not. The same is also true for the opposite. There are eleven types of Interaction Operators outlined in [7], each structuring the messages in a different way. The first four of the Interaction Operators from [7] are used in this thesis, and therefore introduced in Table 3.

Table 3: Subset of Interaction Operators (referenced from [7])[3]

Interaction Operator	Abbreviation	Semantic Description
Alternative	alt	The alternative Interaction Operand depicts a choice of behaviour where at most 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	An option is semantically equivalent to alternative where a choice of behaviour occurs. The default for an option Interaction Operator is one operand, where the either the operand happens, or nothing happens.
Break	-	In a Combined Fragment with the Interaction Operator break, a choice of behaviour is presented where an operand occurs, or the remainder of the interaction is ignored (i.e. termination of system). The operands could be attached to a guard to determine the chosen behaviour. However, a break Interaction Operator without a guard leads to a non-deterministic choice of behaviour.

[3]Interaction operators such as *loop* and *neg* are not a part of the metamodel used in this thesis due to the limitations in the result and is further discussed in Section 8.2.

Parallel	par	The parallel Interaction Operator designates that a parallel merge between all the operands of the Combined Fragment occurs. The order of messages within each operand of the Combined Fragment is preserved. However, the order of messages between operands can be interleaved in any variations.

The Sequence Diagram in Figure 4 also presents a concept of *events*. In a typical Sequence Diagram, events are not labelled, nor are they represented by a 'point' as they are in Figure 4. The representation in Figure 4 is deliberate, to familiarize the users with the concept of events used throughout this thesis. Events are attached to lifelines and denote the sending and receiving of messages. According to [68], there are two rules in the sequencing of events in a Sequence Diagrams;

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.

The authors of [68] also went on to describe that as long as the two rules are followed, ordering the events in a Sequence Diagram is arbitrary. However with reference to the Sequence Diagram metamodel in Figure 3, each *Interaction Fragment* is has a *General Ordering* that has*before*or*after* events in the form of *Event Occurrences*. This ordering framework is adapted for the purpose of this thesis.

2.2 Petri Nets

Petri Net [6] is a formal modelling language that has a strong mathematical foundation and a graphical method of representation. Petri Nets are often used to model control flow in a system and is capable of modelling complex behavioural properties such as *conflicts* (choice) and *concurrencies* (parallelism). The main components of a Petri Net are *places*, *transitions*, *arcs* and *tokens*.

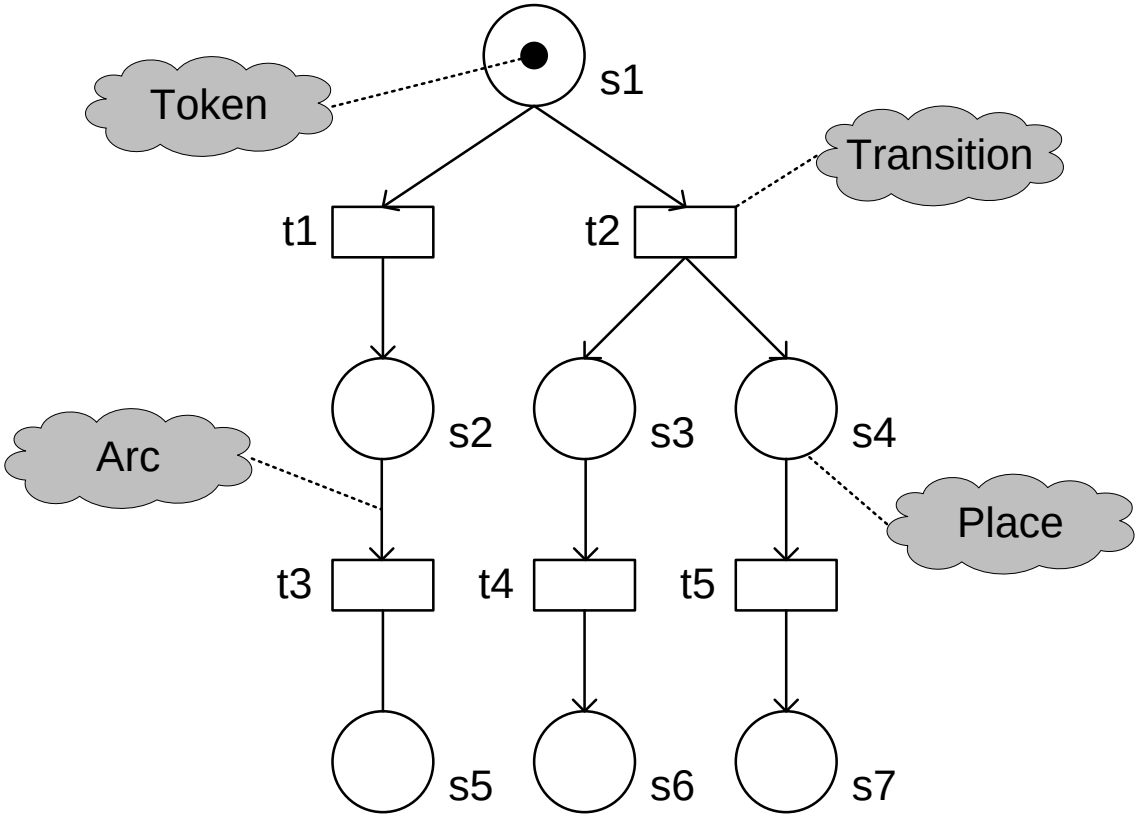

Figure 5: Example of a Petri Net

Figure 5 presents an example of a Petri Net with seven places and five transitions. Each place in a Petri Net may contain a number of tokens, referred to as*mark*. For example in Figure 5, the mark of *s1* is '1' where else for all other places, the mark is '0'. A place is only

allowed to be connected to a transition via either*input arcs* or *output arcs*. An input arc is visually represented as an arrow with the arrow head pointing towards the place (or transition) where else an output arc is an arrow with the arrowhead pointed away from the place (or transition). For a transition to be *enabled* or ready to *fire*, each place that connects to it via an input arc needs to be marked with at least one token. However each token can only be used to fire one transition at a time as would be explained further using Figure 6 which presents an example of a firing sequence for the Petri Net in Figure 5 while highlighting the conflict, concurrency and causal relationships between the nodes.

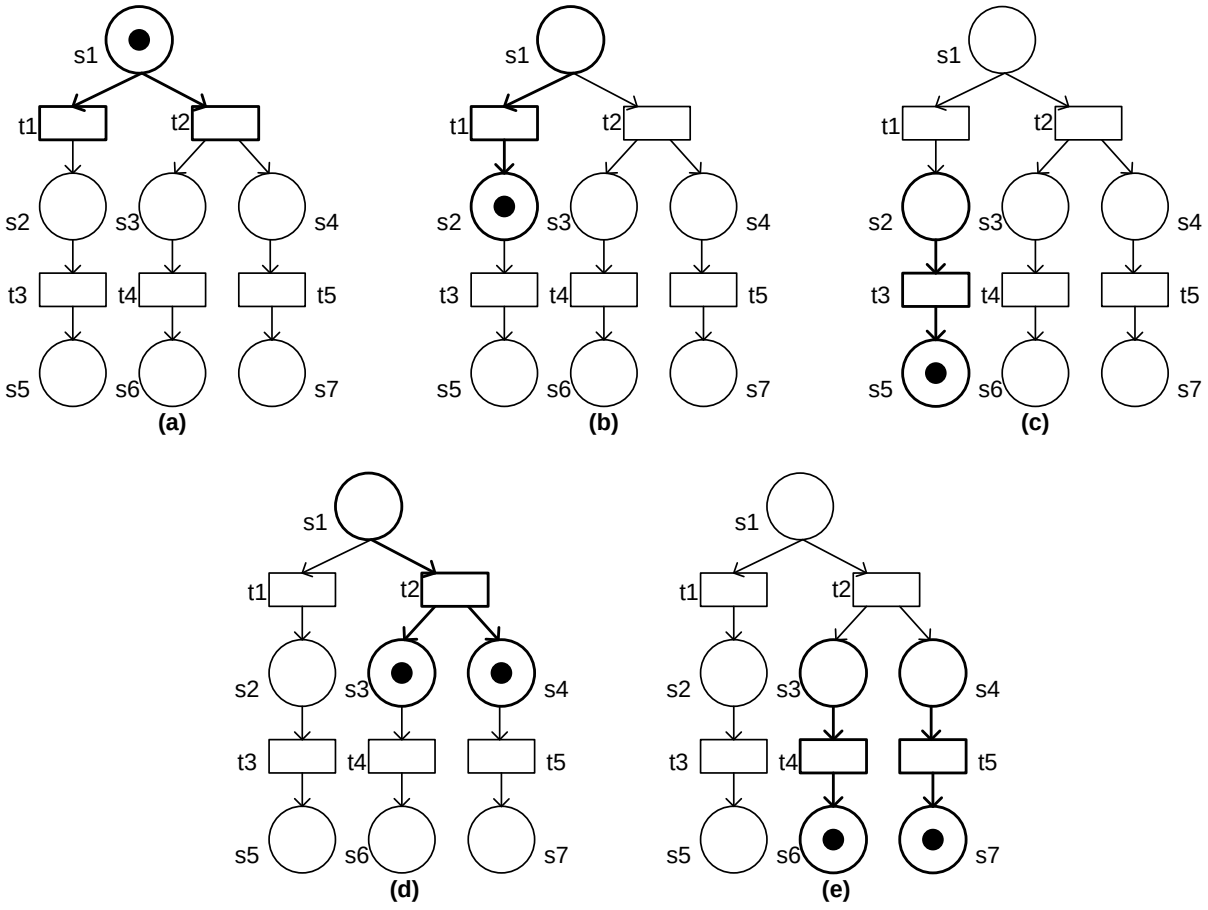

Figure 6: Example of a Petri Net Firing Sequence

Figure 6 (a) shows a marked place, *s1* with output arcs connected to two transitions *t1* and *t2*. Both *t1* and *t2* becomes enabled since every place that is connected to them via input arcs are marked. This represents a conflict where although both *t1* and *t2* are enabled, only one of them may fire, based on the solitary token that is contained in the place *s1*. Figure 6 (b) shows a scenario where *t1* fires. This removes the token from *s1* and places it in *s2* which is the only place connected via an output arc to *t1*. One observation that could be made here is that the previously enabled *t2* is no longer enabled since *s1* no longer contains tokens. Since *t3* is the only transition enabled following the firing of *t1* as depicted in Figure 6 (c), the firing sequence is continued, removing a token from *s2* and placing it in *s5*. The firing of *t3* may only ever occur following the firing of *t1*, thus creating a causal relationship between them. An alternative scenario where *t2* fires instead of *t1* is presented in Figure 6 (d). The firing of *t2*removes a token from *s1* and places a token each in *s3* and *s4*. This is described as a concurrency or parallel relationship where one token is split into two (or more depending on the number of concurrent nodes). The firing sequence then continues with *t4* and *t5* enabled concurrently and independent of each other, as depicted in Figure 6 (e). Another observation that could be made at this point is the absence of any causal relationship between *t4* and *t5* and the order of firing between them is non-deterministic.

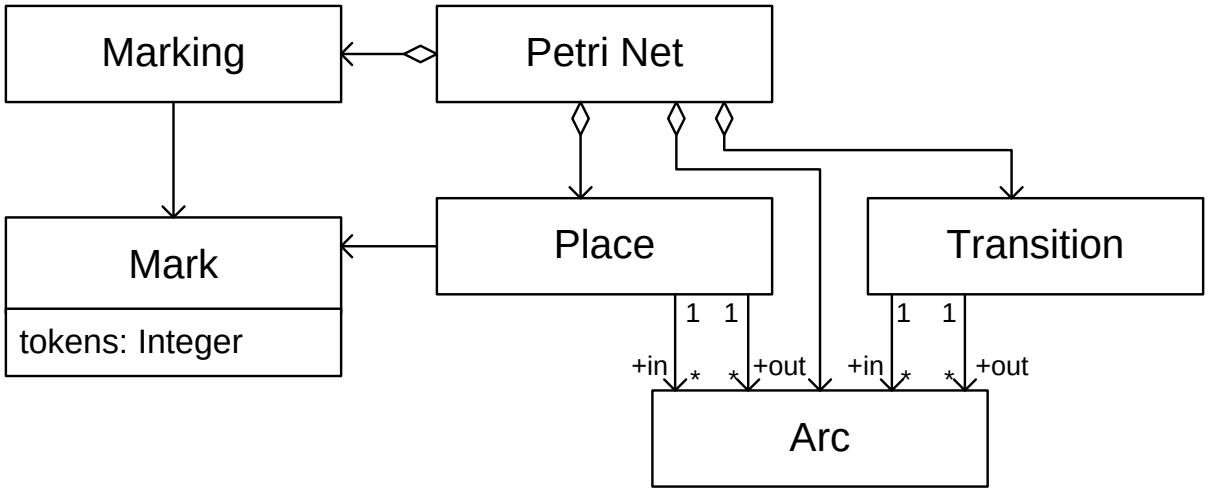

Figure 7: Petri Net Metamodel

Figure 7presents the metamodel of Petri Net where Petri Net consists of at least one place, one transition and one marking. Each place or transition may have any number of input and output arcs and each place has a mark in the form of the integer number of tokens. Alternatively, Petri Nets can also be presented formally as follows:

Definition 1:A Petri Net is a triple $N = (S, T, F)$ where S is a finite set of places and T is a set of transitions where $S \cap T = \varnothing$. F is a relations on $S \cup T$ where $F \cap (S \times S) = F \cap (T \times T) = \varnothing$. 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.

This formalization can also be extended to the relationship between places and transitions. For every place $s \in S$, ${}^{\circ}s$represents the set of transitions that are connected to it via input arcs while s°represents the set of transitions that are connected to it via output arcs. Similarly for each transition $t \in T$, ${}^{\circ}t$represents the set of transitions that are connected to it via input arcs

while t° represents the set of transitions that are connected to it via output arcs. For example in Figure 5, *s1* is an input place for *t1*. This can be presented using the notation $s1 \in {}^{\circ}t1$. The relationship between *s1*, *t1* and *t2* can also be represented as $s1^{\circ} = \{t1, t2\}$.

2.2.1 Flavours of Petri Nets

The mathematical nature of the Petri Net modelling language created a basis for a variety of flavours to be added to the core of the general Petri Net. There have been various extensions made to Petri Nets over the years including the introduction of two types of arcs; the reset arc [69] which resets the place after each termination, making the *reachability* of the net *undecidable*, and the inhibitor arc [6], that allows firing of the transitions only when there are no tokens in the input places of the transition. There have also been variations of the types of Petri Nets such as the Stochastic Petri Nets [70], Coloured Petri Net [71], Prioritized Petri Nets [72] and Dualistic Petri Nets [73].

One of the well-established variations of Petri Nets is the Timed Petri Net. Timed Petri Nets are extensions to the conventional Petri Nets by the inclusion of timing information such as the time associated to the firing of transitions. There are many different interpretations of Timed Petri Nets. However in this thesis, the Timed Petri Net with closed intervals as outlined in [50] are used. The timing information in the metamodel are inferred from the Petri Net tools where [28] shows the existence of two distinct types of transitions; immediate transitions and timed transitions, while[74] states that each time property is modelled via closed intervals. These intervals are defined via specific upper and lower bounds attached to a transition. For a transition to fire, firstly it must be enabled. Secondly, from the moment it gets enabled, a clock starts; the transition can fire when the value of the clock is within the

interval. The inclusion of time constraints in Petri Nets enhances their capability for modelling time-sensitive systems. Moreover, with the benefit of using existing Petri Net tools such as CPNTools [27] and PIPE [28], time related analysis such as performance analysis could take place. Timed Petri Net, its applications and its analysis properties would be further discussed in Chapter 7 of this thesis.

2.2.2 Free Choice Petri Nets

Petri Nets are highly suited for modelling systems with rich, dynamic constructs due to its expressive power and strong mathematical foundation. However as described in [36], analysis algorithms for Petri Nets have a high complexity as a result of its rich modelling capabilities. One possible method to address this issue is to restrict the structural properties of the net, creating a subset of Petri Net called Free Choice Petri Net.

Free Choice Petri Net is a subclass of Petri Nets where conflicts and concurrency could occur, but not simultaneously. This subclass of Petri Net is predominantly used for effective and efficient analysis of a systems [75].

Definition 2: Baccelli [38] defines Free Choice Petri Nets, as whenever two transitions in the net share an input place, they must not have any other input places. This can also be written as when $|s^{\circ}| > 1$, for every $t \in s^{\circ}$, $|{}^{\circ}t| = 1$.

Definition 2 declares that if a place in the Petri Net has more than one output transitions, then each of the transitions must have an input place of exactly one. This ensures that whenever a conflicting behaviour occurs, a concurrent behaviour does not occur simultaneously.

There also exists a weaker definition of Free Choice Petri Net that is also referred to as extended Free Choice Petri Net, as presented in Definition 3.

Definition 3: In a Free Choice Petri Net, if there is an arc from a place *s* to a transition *t*, then there must be an arc from any input place of *t* to any output place of *s*.

However as established in [36], if a Petri Net satisfies the weaker condition in Definition 3, it also satisfies the condition in Definition 2.

The benefits of Free Choice Petri Nets mainly regards to the complexity of various analyses that could be performed on the nets. Esparza and Silva [37] presented an algorithm that allows liveness in bounded Free Choice Petri Nets to be computed in polynomial time as opposed to otherwise exponential complexity of computing liveness in Petri Nets. Rank Theorem [36] also provides a way to compute the liveness and boundedness of a Petri Net in polynomial time by utilising matrix algebra. There are various other studies that are done on Free Choice Petri Nets to reduce the complexity of analysis such as the Reachability Theorem [36], Shortest Sequence Thorem [36], soundness analysis [22], concurrency analysis [76] and the reduction and synthesis rules for Free Choice Petri Nets [36]. In short, Free Choice Petri Nets allow various complex analyses to be conducted in a much less complex manner (polynomial complexity as opposed to exponential complexity) compared to general Petri Nets.

2.2.3 Analysis in Petri Nets

There is a plethora of analysis properties in Petri Nets [6]; however a few well-known properties that are applicable to this research are presented below:

Liveness

Liveness in Petri Nets is commonly associated with the complete absence of deadlock. A Petri Net N is considered *live* if every transition $t \in N$ can be enabled through a *firing sequence* that begins with the transitions enabled at the initial marking M_0. In this scenario, M_0 is often referred to as a *live marking*. By definition, if any transition in N can be enabled through a firing sequence, then every marking M where $M_0 \xrightarrow{\sigma} M$ is also a live marking.

Boundedness

A Petri Net is *bounded* when the number of tokens in every place of the Petri net does not exceed a certain number. Suppose a Petri Net $N = (S, T, F)$ has an initial marking M_0. The Petri Net is regarded as k-bounded if for all $s \in S$, $M(s) \leq k$ where $M_0 \xrightarrow{\sigma} M$. For example, if the number of tokens in each place in N does not exceed one (1) for every marking that results from M_0, the net N is said to be 1-bounded. 1-bounded Petri Nets are also referred to as *safe* nets.

Reachability

Reachability in Petri Nets calculates if a certain state of marking is reachable from the initial marking through a sequence of events. A marking M in a Petri Net N is *reachable* from the initial marking M_0 if there exist a firing sequence σ such that $M_0 \xrightarrow{\sigma} M$.

Reversibility

Reversability analysis can be described as complementary to the reachability analysis. Suppose a Petri Net N with the initial marking M_0 and a set of markings $M \in \mathcal{M}$. The Petri Net N is *reversible* if for every marking $M \in \mathcal{M}$, there exist a firing sequence σ such that $M \xrightarrow{\sigma} M_0$.

Persistence

Persistence analysis in Petri Net refers to the firing of two or more enabled transitions under a single marking. Suppose a set of enabled transitions T_e in a Petri Net N with under the marking M. If the Petri Net N is *persistent*, then the firing of any single transition $t \in T_e$ would not *disable* any other transitions in T_e. This reflects concurrency or parallelism where every enabled transition remains enabled until it fires. A persistent net also belongs to a class of Petri Nets called *Marked Graph*s, a subset of Free Choice Petri Nets where conflicting behaviours could not occur.

These properties and more are analysed using three main analysis methods in Petri Nets: Reachability Tree, State Equation or Incidence Matrix and the Reduction method. The Reachability Tree approach involves the enumeration of all reachable markings in the Petri Net. However, this approach is limited to smaller sized Petri Nets due to the complexity of the *state-space explosion* orcalculating reachable states for each marking in the net. The equation and reduction methods on the other hand are more powerful and allow analysis to be performed on larger nets. The State Equation or Incidence Matrix method performs algebraic analysis on Petri Net behaviours that are expressed as matrices and equations. Meanwhile the Reduction method is used to reduce large scale Petri Nets into smaller nets while preserving the system properties to be analysed. However by using Petri Net tools, the complexity behind these methods are hidden from the users and as such are not explained. Mathematical explanation of the methods are found at [6].

2.2.4 Petri Net Tools

One of the main attractions of Petri Nets is the plethora of widely available tools to design, share and analyse Petri Net models. The list of Petri Net tools are available in a comprehensive tool database that is updated regularly at [34]. In Table 4, a selection of Petri Net tools and their properties are presented.

Table 4: Petri Net Tools

Petri Net Tool	Modelling		Type		Analysis			Platform Independent	Free of Charge
	Graphical Editor	Token-Game Animation	Timed Petri Nets	High Level Petri Nets	Structural Analysis	Behavioural Analysis	Performance Analysis		
AlPiNA [29]	•			•	•	•	•	•	•
COSA BPM [77]	•	•		•				•	
CPNTools [27]	•	•	•	•	•	•	•	•	•
GreatSPN [30]*	•	•	•	•	•	•	•		•
Helena [78]				•	•				•
HPSim [79]	•	•	•				•		•
INA [80]			•	•	•	•	•	•	•
JFern [31]	•	•	•	•			•	•	•
LoLA [32]				•	•			•	•
Maria [81]	•	•		•	•	•		•	•
NetLab [33]*	•	•			•				•
PetriSim [82]	•		•	•					•
PIPE [28]	•	•	•		•	•	•	•	•
TimeNet [83]*	•	•	•	•	•		•		•
Tina [35]	•	•	•		•	•		•	•

Table 4 presents a comparison between fifteen Petri Net tools based on nine evaluation criteria. The first group of criteria – modelling, judges if a Petri Net tool supports visual modelling of the Petri Nets and if it able to present an animation of the token-game (the flow of token in the Petri Net). Majority of the tools provide a platform for visually modelling the Petri Nets and a token-game animation. However, there are also tools (i.e. Helena, INA and LoLA) that only accept text-based modelling of the Petri Nets.

The second group of criteria that is evaluated in Table 4 is the type of Petri Nets supported by the tool. Although there are various types of Petri Nets, only Timed Petri Nets and High Level Petri Nets (i.e. Coloured Petri Nets) have bearing towards this research project. As such, only the two types of Petri Nets are considered. Following the types of Petri Nets, the next group of criteria is to evaluate the types of analysis that could be performed by the Petri Net tools. Three types of analysis are evaluated; structural analysis (i.e. liveness analysis, boundedness analysis), behavioural analysis (i.e. reachability analysis) and performance analysis (i.e. throughput analysis, waiting time analysis). This is followed by the platform independence criteria which evaluates if a tool could function across all the major platforms (i.e. Windows, Linux, Apple, Sun, etc). Finally, the cost of obtaining the tool is considered where most of the tools could be obtained free of charge, including the three tools that are marked '*' which are free of charge for academic purposes.

In this research project, two of the tools presented in Table 4 are used for modelling and analysis of Petri Nets; CPNTools and PIPE. Both CPNTools and PIPE are capable in modelling the Petri Net visually as well as animating the flow of tokens in the system. Both the tools could also be used to perform all the types of analysis evaluated in Table 4. Both CPNTools and PIPE can be used across various platforms and can be obtained free of charge. CPNTools is also highly capable in modelling Timed Petri Nets and High Level Petri nets. However, PIPE is not suitable in modelling High Level Petri Nets. Nonetheless this limitation is overshadowed by its other qualities as presented in Table 4.

2.3 Labelled Event Structures

Event Structures [42] are models of computational process that allows a system to be modelled as a sequence of events. Event Structure models the behaviour of a system through the relationship between the various events in the system. There are three main types of relationship between events; causal relationship, conflicting relationship and concurrent relationship.

Definition 4: An Event Structure is a triple, $E = (Ev, \rightarrow^*, \#)$ where Ev is a set of events and $\rightarrow^*$ and # representing binary relations *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 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 5: 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 have 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 could also be written as $e_1 \rightarrow e_2$.

Definition 6: Let E= (Ev, $\rightarrow$*, #) be a Discrete Event Structure and L an arbitrary set where l:$Ev \rightarrow L$ would be the labeling 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 next section presents the translation from Sequence Diagrams to LES followed by the unfolding of Petri Nets into LES.

2.3.1 Translating UML Sequence Diagrams into Labelled Event Structures

In this section, a translation of Sequence Diagrams into LES is presented based on the semantics in [40]. In order for a Sequence Diagram to be represented as an LES, 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 presented in Definitions 7, 8 and 9 respectively.

Definition 7: A Sequence Diagram can be represented as a tuple SD = (I, Loc, Loc_{ini}, Mes, E, $Path$, X_I) where:

- I is the 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
- $\{X_i\}$ 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

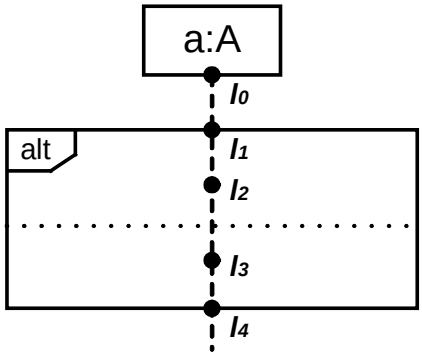

Figure 8: Example of events in a Sequence Diagram

Definition 8: *Scope* is a function given by *scope*: *Loc*→*Path*. Referring to Figure 8above, *scope*(l_2) = alt(2)#1 and *scope*(l_3) = alt(2)#2. This can be 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 signifies 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 9:*Alt_occ* is a local function given by *alt_occ*: *loc*(*i*) →ℕthat returns a possible number of alternative scenarios that can lead to a specific location. Referring to Figure 8 above, *alt_occ*(l_4) = 2 because there are 2 possible scenarios that could lead to l_4 from the initial location l_0 which are scenarios $S_1 = \{l_0, l_1, l_2, l_4\}$ and $S_2 = \{l_0, l_1, l_3, l_4\}$.

By using the local function*scope*, messages that are not causal and have a relationship of either *conflict* or *concurrent* can be identified. This information would be essential in the creation of the LES. On the other hand, by using the local function*alt_occ*, the number of

alternative scenarios that leads to a specific location in the diagram can be obtained, as to create the branches in the corresponding LES.

Following the example in Figure 8above, a fragment of LES that corresponds to that particular Sequence Diagram can be created. Since l_4 has an *alt_occ* of 2, then it has two events associated to it; e_4 and e_5. The rest of the locations have an *alt_occ* of one, and will be represented by e_1, e_2, and e_3 respectively. Therefore, with 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 Figure 9.

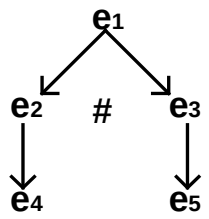

Figure 9: Labelled Event Structure corresponding to the Sequence Diagram in Figure 8

An example of translating Sequence Diagrams into LES is presented in Appendix A.

2.3.2 Unfolding Petri Nets into Labelled Event Structures

In this section, a method that maps Petri Nets into LES is presented based on a branching process of Petri Nets called unfolding. This method, introduced by McMillan [41] creates a net where nodes in the net 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 sometimes

referred to as a Labelled Causal Net or a Labelled Occurrence Net and can be interpreted as a Labelled Event Structure.

Definition 10: Referring to [41], suppose a Petri Net $N = (S, T, F)$, then a *Labelled Occurrence Net* (unfolding of N) consist of a Petri Net $N' = (S', T', F')$ and a labelling function L' which maps P' onto the set P and T' onto the set 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 $p \in P'$, $p \in t_1^\bullet$ and $p \in t_2^\bullet$, then t_1 and t_2 must ne the same.
- No self-conflicts: if $t_1, t_2, t_3 \in T'$, $t_1 F'^* t_3$, $t_2 F'^* t_3$ and ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \varnothing$, 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 starts with the generation 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 better understanding of how a Petri Net is unfolded into LES, an example of the process is presented in Appendix A.

2.4 Model Driven Development

Model Driven Development [84] aims to promote the role of *modelling* in software development. Models in the context of MDD are captured in machine-readable representations, using languages which are widely adopted by software industry [7]. Hence it is possible to communicate such models to various parties and reuse them. This results in lower software production cost and shorter development cycles. For the purpose of this thesis, MDD is used in the seamless transition of models between two languages; Sequence Diagrams and Petri Nets. As such, software system design can be conducted in Sequence Diagrams while the more formal notions of analysis could be performed in Petri Nets.

The transitions of models as used in this thesis adopts the standards set by Model Driven Architecture (MDA) [85], a flavour of MDD which is initiated by the Object Management Group (OMG). MDA outlines the concept of *model transformation* which is central to the work presented in this thesis. Another standard that is central to the concept of model transformation is Meta Object Facility (MOF) [86],used for describing *metamodels*. Metamodels are themselves *models*, from which models of the system are instantiated. MOF can be compared to EBNF, which is used for defining programming languages grammars. As a result, MOF is a blueprint from which *MOF Compliant metamodels* are created.

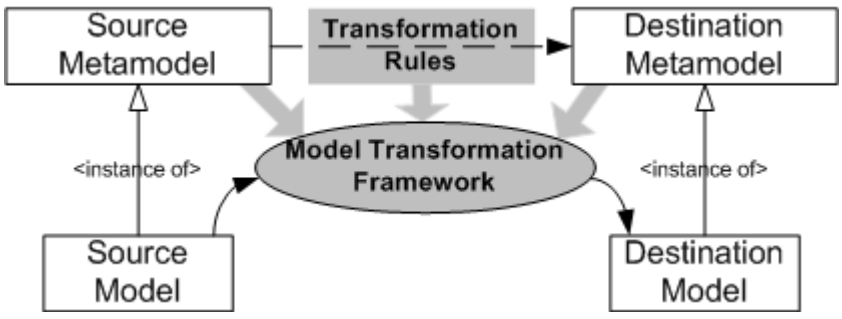

Figure 10: Model Driven Development Model Transformation

Figure 10depicts an sketch of MDD model transformation as outlined by MDA [85] and the metamodels that comply to MOF [86]. A number of *transformationrules* are used to define how various elements of one metamodel (*source metamodel*) are mapped into the elements of another metamodel (*destination metamodel*). The process of model transformation is carried out automatically via the software tools which are commonly referred to as *model transformation frameworks* [87-89]. A typical model transformation framework requires three inputs: source metamodel, destination metamodel and Transformation Rules. For any instance of the source metamodel, a *transformation engine* executes the rules to create an instance of the destination metamodel.

CHAPTER 3

MULTI PARADIGM MODELLING

This chapter discusses the methodology used behind this research, Multi Paradigm Modelling and how it could be used to solve the problem statement presented earlier in this thesis.

3.1 Role of Modelling in System Development

Modelling is becoming an essential component of system development. A model, according to Blaha and Rumbaugh [90], is defined as an abstraction of something, used for the purpose of understanding it before it is built. By applying this definition in the context of system development, a model can be defined as an abstraction of the system that is used to understand the works of the system before it is built. As such, modelling affords a system designer the luxury of visualizing the system before it is developed.

At present, modelling plays an important part in system design. Van Gigch [91] states that there are three domains involved in system design; *reality*, modelling and *metamodelling*.

Reality represents the notion of the system in real life where else modelling represents an abstraction of the reality by translating it into a verbal, graphical or mathematical notation. Metamodelling on the other hand represents an abstraction of modelling or the modelling of the modelling process. In short, the role of modelling in system design can be explained as the process of translating a real life view of the system into a verbal, graphical or mathematical notation, based on the metamodel.

As well as the different levels of abstraction, modelling may involve different levels of formalisms. Three levels of formalisms are identified in [92]; natural language, semi-formal notation and formal notation. Natural language models are highly expressive and flexible, as they contain descriptions and annotations that are easy to read. However, due to the lack of semantics, the models could be interpreted differently by each stakeholder. On the other hand, models with semi-formal notation use notational semantics to express the structure or behaviour of the system. An example of this type of model is Unified Modelling Language (UML) [7]. Finally, models with formal notations are known for their precise semantics with underlying mathematical structures. For example, modelling languages with formal notations are among others Z [4], Alloy [5] and Petri Nets[6]. This type of model is commonly used for extensive reasoning and model analysis due to its mathematical nature.

The role of modelling in system development is often confined to just model design. Although it is undeniable that designing a model is essential, the role of modelling need not end there. The system models could also be subjected to model analysis to evaluate the structural rigidity and the behavioural properties of the system, or model synthesis where two or more models with common elements could be put together in order to get a more holistic view of the system. The following sections present model design, model analysis and model

synthesis; discussing their definitions and their importance in the role of modelling for system development.

3.1.1 Model Design

Model design is the process of representing a view of the system in the form of models. For this purpose, system designers often opt for a semi-formal notational model as it provides the best balance between ease-of-use and precision. As a result, UML [7] has become the preferred language in model design.

There are various types of model in UML, divided mainly into structural diagrams and behavioural diagrams. Structural diagrams such as Class Diagrams and Object Diagrams represent the composition of the system, describing the various elements that make-up the system. Behavioural diagrams such as Activity Diagram and Sequence Diagram on the other hand describe the behaviour of the system under different circumstances as well as the interaction between elements in the system.

The different types of models, as well as the well established set of semantics for each model type in [7] allows system designers to accurately project their views of the system into models. The easily comprehendible nature of UML models also allows for straightforward communication between stakeholders of the system without extensive knowledge of any particular modelling or programming language.

3.1.2 Model Analysis

Model analysis could be regarded as a preliminary analysis of the system. Performing mathematical analysis on models of the system could provide crucial feedback on any

structural design flaws or unwanted behaviours in the system. This allows the system designer to rectify the design faults even before the system is built, essentially saving countless man-hours and resources from having to re-build the systems.

The mathematical nature of the analysis process dictates the need for modelling languages with formal semantics such as Alloy and Petri Nets. Alloy is a declarative language that models a system based on first-order logic [93]. It is highly suited to perform structural analysis of a system. Petri Nets on the other hand is a state-based modelling language that is capable of performing various types of performance analysis. The commonality between Alloy and Petri Nets, as well as other formal modelling languages such as B and Z is the strong mathematical foundation behind the language that makes it suitable for precise computational analysis.

Using model analysis, crucial mistakes can be avoided in the system development process. Various types of analysis, such as liveness analysis, deadlock detection and boundedness analysis could all be performed to effectively void the system of unwanted behaviour. Examples of analysis in Petri Nets as well as their relevance to system development are explained in Section 2.2.3. The importance of model analysis is also highlighted in [94], among them are the computation of dependencies between states as well as a risk analysis.

3.1.3 Model Synthesis

Model synthesis or model composition is the process of allowing two or more models to be put together based on a set of common elements. This is necessary as modern complex system must be broken down in the design phase, where each module of the system is designed separately and independently of each other to reduce the overall complexity of the model.

There are also cases where models are built based on specific scenarios or from a particularperspective such as security or quality-of-service (QoS). Performing model synthesis between the different modules or integrating the various perspectives of a system could not only present an integrated view of the system; it also highlights the dependencies between the various modules and viewpoints.

Model synthesis could also be applied to more enterprise systems in the form of *plug-ins*. There exists a concept called *refinement* in model synthesis where a set of behaviours could be plugged into an existing model without having to redesign the whole model. For example, in designing a secure system, a system designer could plug in various security protocols into the system design to find the best fit for needs of the system without having to create multiple models.

The notion of model synthesis is well-established in some modelling languages such as Petri Nets [12, 16, 17, 21-23, 26, 58, 59] where various techniques and algorithms exist for different types of synthesis. For example, refinement of a particular state in the Petri Net calls for a *top-down*[22] synthesis method using a *place refinement* or *transition refinement* algorithm [16]. On the other hand, synthesis of two Petri Nets from different perspectives can be performed using a *bottom-up* approach where the common elements between the Petri Nets are merged, causing the integration of the nets.

3.2 Bridging the Gap between Design, Analysis and Synthesis of Models

As established in the previous section, the role of modelling in system development extends not only to the model design, but also to model analysis and model synthesis. However, the different requirements for the design, analysis and synthesis phases lead to the use of different modelling languages. This results in *heterogeneity* where model design is often performed in a semi-formal, easy to use language such as UML while model analysis and synthesis are conducted on a more mathematical formal language such as Petri Nets. The incompatible nature of the heterogeneous models compounded by the lack of interoperability between the toolsets of the languages present a serious challenge to system developers [8, 75, 95]; to provide a platform that allows interoperability between models with different levels of formalisms. One possible solution for this challenge is Multi Paradigm Modelling.

3.2.1 Introduction of Multi Paradigm Modelling

Multi Paradigm Modelling[96-103] is a platform that promotes interoperability between heterogeneous models. Applying Multi Paradigm Modelling in modelling and simulation [99-102], Vangheluwe et al [103] described it as a field that addresses three directions of research; multi-formalism modelling, model abstraction and metamodelling.

3.2.1.1 Multi-Formalism Modelling

Multi-formalism modelling provides an interoperability platform for models with differing levels of formalisms on the basis of model transformation. Model transformation is the process of translating one model into another using a set of predetermined rules.

Currently, model transformation plays a key role in Model Driven Development (MDD) [85]. According to a survey on model transformation [104], the intended application of model transformation include generating low-level models from higher level models, synchronizing models with different levels of formalisms and reverse engineering higher level models from low-level models. There are various frameworks available for model transformation, among others VIATRA (Visual Automated model Transformations)[105, 106], Kent Model Transformation Language [107], ATL [108], Kermeta [109] and SiTra [110, 111]. A common way to express a model transformation is using QVT relational language [112]. QVT is a standard for model transformation defined by Object Management Group (OMG).

A few key features that are common to all model transformation as described in [104] include *specification*, such as the *pre* and *post* conditions for a model transformation, the set of transformation rules, the directionality of the transformation as well as the source and target relationship. In an MDD model transformation, a source metamodel and a target metamodel are also required, whereby each source and target model should conform to the respective metamodels.

3.2.1.2 Model Abstraction

Model abstraction is the process of removing a certain low-level detail from the model while preserving the construct and general behaviour of the system. Similarly to multi-formalism modelling, model abstraction also uses model transformation. However, a significant difference between the two model transformations is that for model abstraction, the source and destination models are of the level of formalism.

Model abstraction is often used in removing various complicated low-level behaviours in the system, according the requirements of a specific perspective. For example, a complete model of the system filled with low-level behaviour might be too complicated for distribution to various stakeholders. However using model abstraction, the model could be simplified up to a certain level without losing its structural properties and vital behaviours. The same concept can also be used for optimization [101] of models. Using a base model that is filled with all the details, less detailed models can be automatically derived from it for various operation tasks such as control design and performance assessment.

3.2.1.3 Metamodelling in Multi-Paradigm Modelling

Metamodelling (as explained in Chapter 2) refers to the modelling of models. Metamodel or *model of models* is itself a model that defines other models. For 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 $\mathbb{M}_{\mathcal{L}}$.

Mosterman and Vangheluwe[101] describe the advantages of metamodelling as numerous. The metamodel of a modelling language can be regarded as a specification for the language which can either be used for documentation purposes or as a basis for model

analysis. Metamodelling also allows new languages to be born just by modifying or tweaking parts of existing metamodels. This allows customization of the modelling languages to serve a specific purpose.

3.2.2 Review of Existing Work

Multi Paradigm Modelling is an active area of research where many studies are being conducted. Among others, Vangheluwe et al [103] presents an approach where Multi Paradigm Modelling is applied to modelling and simulation [99-102] while Henkler and Hirsch [113] apply Multi Paradigm Modelling to reconfigurable mechatronic systems by allowing Mechatronic UML [114] to perform verification and code generation. However in this section, the focus of the existing work review is on bridging the gap between design and analysis, as well as design and synthesis.

3.2.2.1 Design and Analysis

Bridging the gap between the model design phase and the model analysis phase in system development is vital due to the varying levels of formalisms between the models involved. System design is often performed in UML while the model analysis is commonly performed in a more formal mathematical language.

One such work is UML2Alloy [8, 115] by an alumnus of University of Birmingham, Kyriakos Anastasakis. UML2Alloy is a tool that allows UML models to be analysed using Alloy. The implementation of UML2Alloy was created using an MDD model transformation that transformed UML Class Diagrams [7] augmented with OCL [116] constraints into Alloy models. The Alloy model is then analysed using Alloy Analyzer, a tool that allows model level analysis using first order logic. UML2Alloy was developed on the platform of Java

using SiTra [110] as the model transformation framework. Although Alloy is highly suited to model static models and constraints, it has its limitations when it comes to dynamic behavioural models. Although some dynamic properties could be modelled using *pre* and *post* conditions, Alloy does not have the mechanism to model complex behaviours such as parallelism.

Besides UML2Alloy, there have also been other model transformation approaches to bridging the gap between model design and analysis. Kim [117] presents a model transformation from both Class Diagrams and State Machines into Object-Z using MDA technology. However, to the best of my knowledge, this transformation has not yet been implemented. A similar approach is also adopted in [118] and [119] where Class Diagrams and OCL Constraints are transformed into the formal language B [120]. In particular, [119] proposes a UML profile for B called UML-B and the automation of the transformation with a tool called U2B. A major feature of this approach is that it makes use of B provers to check the conformance of the operations' pre and post conditions to the invariants of the model. The main difficulty with provers, as underlined in [119], is that even semi-automatic provers assume a substantial amount of knowledge from the user.

Another school of thinking to solve the heterogeneity problem between design and analysis models is the integration of formal method techniques into UML [117-119, 121, 122]. Using this method, a formal and mathematical semantics adopted from a formal language is integrated into UML to allow analysis to be performed without transforming the individual models. Examples of UML formalization include Evans *et al*[121], who propose the use of Z as the underlying semantics for Class Diagrams to deal with the static aspects of models and Küster-Filipe [40], who presents a mathematical semantics for Sequence Diagrams based on Labelled Event Structures.

Among the advantages of formalization include including the ability to analyse a model via techniques such as model checking and theorem proving in order to ensure correct specification. The introduction of logical and timing constraints into a model, in particular, facilitates the investigation of non-functional aspects of the system such as QoS and security. However it is worth noting that formalization comes at a cost – simplicity. The main reason UML is chosen as the language for model design is its simplicity and semi-formal semantics. Formalization creates a formal semantics for UML, making it harder to use and thus reducing its appeal.

3.2.2.2 Design and Synthesis

The disparity between design models and models that can be synthesized stems from the lack of synthesis capabilities in UML. Therefore, in order for synthesis to be carried out in UML models, the Multi Paradigm Modelling platform is invoked to either transform the UML models into a more synthesis-friendly language, or adopt the notion of synthesis form a different language into UML.

Liang et al. [43] describe a method for synthesis or integration of Sequence Diagrams based on formalization of the Sequence Diagram into typed graphs. This method however is designed for Sequence Diagrams that only consist of messages and lifelines without any complex constructs such as parallelisms and conflicts as evident in page 133 of [44]. Bowles and Bordbar [45] on the other hand present a method of synthesis by mapping a design consisting of multiple views modelled by Sequence Diagrams into a unique mathematical model which is used for analysis and detecting inconsistencies. This approach can also be viewed as an instance of multi-formalism modelling where the Sequence Diagrams are essentially transformed into a mathematical model before the synthesis is performed. Krüger

[47] also presents an approach for synthesis of Message Sequence Charts (MSC) [123] where a notational semantics for MSC is introduced. A notion of message refinement is also introduced in [47], where a message is syntactically replaced by a protocol for every occurrence in the MSC. However, this approach of refinement does not preserve the equivalence relations as stated in page 172 of [47]. A more primitive method of synthesis can also be performed in UML – manual synthesis. However as evident in [8], synthesis of non-trivial models could be tedious and redundant.

3.2.3 Using Multi Paradigm Modelling to Bridge the Gap between Design, Analysis and Synthesis of Models

It has been previously established that Multi Paradigm Modelling is highly suited in bridging the gap between heterogeneous models. In this section, an introduction to my research is presented – how Multi Paradigm Modelling is used to bridge the gap between model designs in Sequence Diagrams, model analysis in Petri Nets, and adopting the notion of model synthesis from Petri Nets into Sequence Diagrams. On top of that, a brief introduction is also presented on how semantic preservation between the heterogeneous models can be established using Multi Paradigm Modelling.

3.2.3.1 Model Design Language

One of the primary goals of this research is to extend the capabilities of UML2Alloy in bridging the gap between model design and analysis into dynamic behavioural models. As such, UML Sequence Diagram is chosen as the language for designing the model due to its capabilities in modelling complex behavioural properties and interactions.

3.2.3.2 Model Design to Model Analysis

Similar to UML2Alloy, a formal mathematical modelling language is chosen to perform model analysis. In this research, the formal language of choice is Petri Nets due to its capability to model dynamic behavioural models, its extensive capacity in model analysis as well as a strong research community.

Using Multi Paradigm Modelling as a platform, the Sequence Diagrams from the system design could be transformed into Petri Nets using a set of transformation rules. This creates model interoperability [60, 124, 125] between the Sequence Diagram models and Petri Net models. This interoperability presents the system designer a chance to perform model analysis before the system is built.

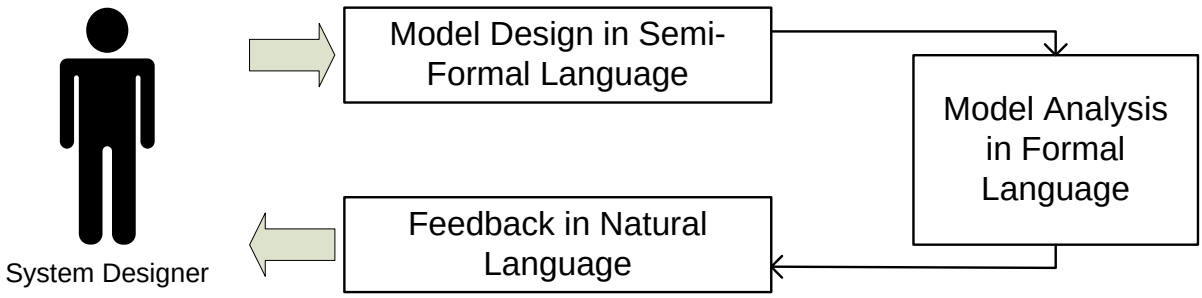

Figure 11: Example of Model Design and Analysis via Multi Paradigm Modelling

Figure 11 presents a scenario in Multi Paradigm Modelling between three levels of formalisms, where a system designer models a system in a semi-formal language, performs analysis on the model using a formal language and receives the feedback in natural language. An example of this application is when a system designer takes advantage of the easy-to-use, rich constructs of UML Sequence Diagrams to design the model. The interoperability platform provided by Multi Paradigm Modelling allows the models designed in Sequence Diagrams to then be analyzed in Petri Nets; a more formal, mathematical language. By using

the automated model transformation within Multi Paradigm Modelling, the system designer does not even need extensive knowledge of Petri Nets or Petri Net tools to perform the model analysis. Finally, the analysis result can be obtained in the form of reports from the Petri Net tools. This scenario in this example clearly illustrates the advantages of using Multi Paradigm Modelling in model analysis.

The model transformation from Sequence Diagram to Petri Nets is described in Chapter 4 while the use of Petri Nets foundations for analysis of Sequence Diagrams is presented in Chapter 5.

3.2.3.3 Model Design to Model Synthesis

The previous sections have established that Sequence Diagram is the language of choice for model design and Petri Nets the choice for model analysis. Using the same combination, the well-studied notion of synthesis in Petri Nets can also be adapted in Sequence Diagrams.

One way to perform synthesis for Sequence Diagram models would to take advantage of the model transformation platform provided by Multi Paradigm Modelling to transform the Sequence Diagrams into Petri Nets, perform the synthesis, and transform the Petri Nets back into Sequence Diagrams. However, bi-directionality in this model transformation is still a subject for future research, thus it is not currently possible.

Another method to perform synthesis in Sequence Diagrams is to adopt the well-established notions of synthesis in Petri Nets into Sequence Diagrams. This can be done by studying the equivalence relation between the Sequence Diagrams and Petri Nets through the transformation rules and amend the algorithms accordingly. This is depicted in Figure 12 where a synthesis technique from one paradigm is adapted for use in a different paradigm.

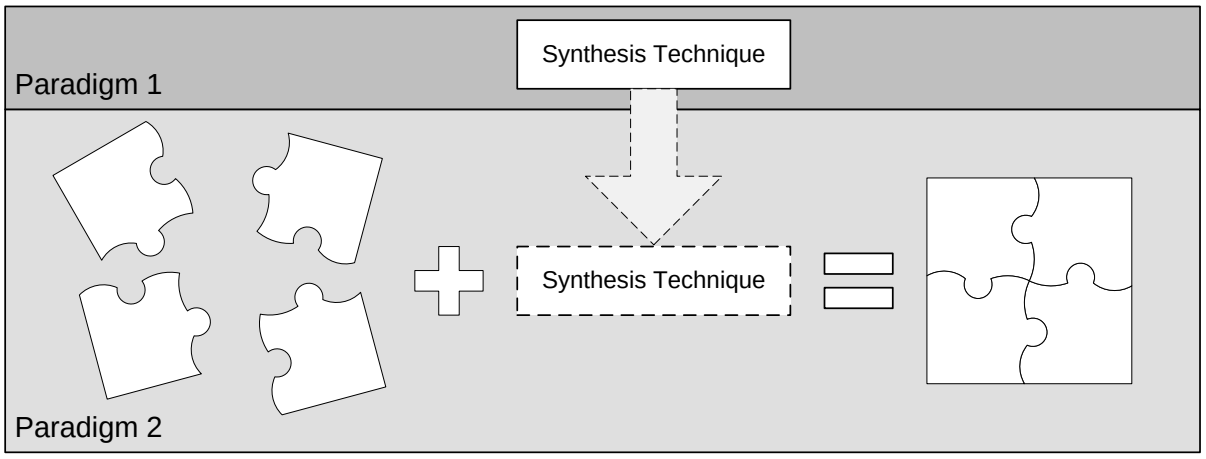

Figure 12: Example of Model Synthesis via Multi Paradigm Modelling

The scenario in Figure 12illustrates that Multi Paradigm Modelling not only allows models to be transformed from one language to another; it also allows constructs and methods to be adapted from one paradigm to the other. A more complete description of Sequence Diagram synthesis using Petri Net foundations is presented in Chapter 6.

3.2.3.4 Semantics Preservation in Multi Paradigm Modelling

By now, it has been made clear that Multi Paradigm Modelling provides a platform for interoperability between heterogeneous models. However proving that the model transformation does not alter the semantics of the original model is a different area of research altogether.Interest in this area does exist, where in a case-study featuring the model transformation of two simple, self-defined languages[126], the authors presented a discussion of two techniques to proof the correctness of their model transformation; using triple-graph grammar and in-situ transformation. However as of yet, this approach has not been applied to a real-life model transformation between two complex languages. Sousa [127] on the other hand, introduced a real-life model transformation from a Domain Specific Language to automatically create Complex Control Systems Graphical User Interfaces and claimed that the

semantics are preserved; through manually comparing the *paths* between the source and destination languages. There has also been an approach in [128] where a method for proving correctness using a mathematical formalism called Constructive Type Theory (CTT) is introduced where every model, metamodel, model transformation are all presented uniformly in CTT. This approach is also extended in [129] by using Calculus of Inductive Constructions. Other approaches to present semantic equivalence in model transformation include [130], which is not dissimilar to the approaches in [128, 129]; as well as [131] which used an automatic *state-space checker*to compare the states of the source and destination model; and [132] which presents semantic preservation through informal observation of test-case in each step of the model transformation.

The sketch of a proof of *correctness* for the model transformation in this thesis is presented in [62]where the semantic preservationof themodel transformation is proven using a common semantic domain. Suppose a model transformation from the source model to the destination model in two different paradigms. The correctness of this model transformation could be proven by the introduction of a third paradigm, provided both the source and destination models could be translated into the third paradigm. Figure 13 illustrates this scenario.

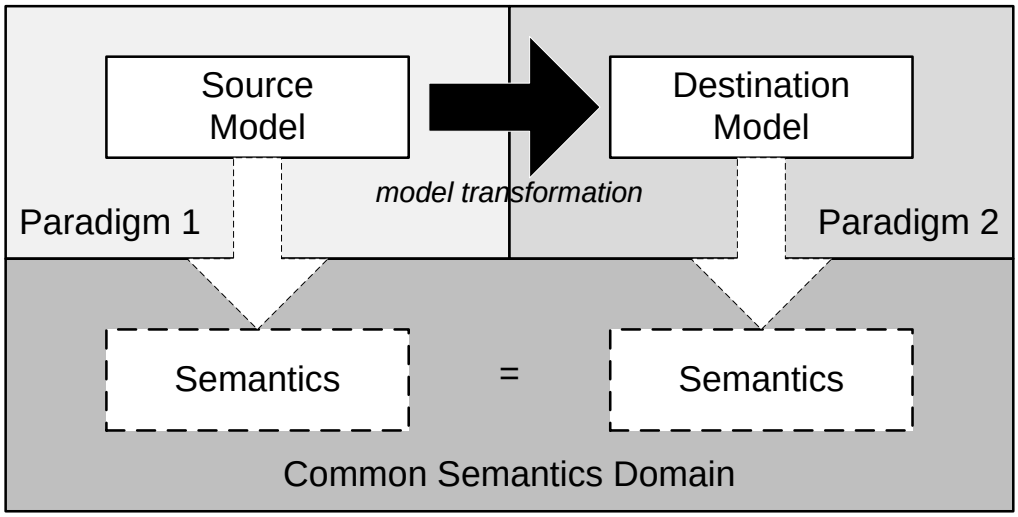

Figure 13: Using a Common Semantics Domain to Prove Correctness of Model Transformation

In the case of this research, the correctness of the model transformation from Sequence Diagram to Petri Nets is proven using a common semantics in Labelled Event Structures (LES); a graph that could represent the semantics of both Sequence Diagrams and Petri Nets. By comparing the LES generated by fro both paradigms, the correctness of the model transformation can be established. A more complete view of this proof is presented in Chapter 4.

CHAPTER 4

SD2PN – SEQUENCE DIAGRAMS TO PETRI NETS

Sequence Diagram, a member of the Unified Modelling Language (UML) family is an interaction based modelling language that describes a system as a flow of events between objects. Petri Net on the other hand is a formal, mathematical language that is typically used for various types of analysis. Petri Net is also highly capable of modelling the flow of events in a system. This commonality between the languages, compounded with the vastly different levels of formalism and abstraction between Sequence Diagrams and Petri Nets, makes them a candidate for implementing Multi Paradigm Modelling. The user friendly, low-formalism Sequence Diagram could provide a platform for designing the system as well as communicating the system design with other stakeholders, where else the mathematical nature of Petri Nets could be used for analysis and manipulating the more formal elements of the system.

In this chapter, an MDD model transformation, SD2PN [62] is introduced. SD2PN provides a framework for Sequence Diagrams to be transformed into Petri Nets, and serves as

a basis for the Multi Paradigm Modelling. This is followed by a proof that all Petri Nets generated by SD2PN are Free Choice Petri Nets, a very well-studied subclass of Petri Nets. This discovery and its significance are presented in Section 4.2. Finally, the correctness of the model transformation is established in Section 4.3.

4.1 SD2PN – The Model Transformation

SD2PN is a rule-based MDD model transformation that transforms any Sequence Diagrams that conforms to the metamodel in Chapter 2 into Petri Nets. The model transformation process is hereby described in three stages:

Stage 1: **Decomposition**

The Sequence Diagram inputted into SD2PN is decomposed into multiple small fragments based on the Sequence Diagram metamodel.

Stage 2: **Transformation**

Each Sequence Diagram fragment from Stage 1 is transformed into a Petri Net block based on a set of model transformation rules.

Stage 3: **Composition**

The Petri Net blocks from Stage 2 are put together using two local functions; *morph* and *substitute*.

Each instance of the model transformation goes through the three stages in order to successfully transform Sequence Diagrams into Petri Nets. The three stages are explained in depth in Sections 4.1.1, 4.1.2 and 4.1.3.

4.1.1 Decomposition

The process of decomposition of a Sequence Diagram is carried out on the concrete syntax representation and involves identification of various model elements and their relationships. The metamodel of Figure 3in Chapter 2depicts model elements used in a Sequence Diagram.

The main model element chosen from the metamodel is *message*. *Message* refers to the events, or the flow of information between objects in the Sequence Diagrams. Each *message* consists of two *MessageEnds*, as its sending and receiving events. These *MessageEnds* are instances of *EventOccurrence*; where the causality of the events is determined by *GeneralOrdering*. In Sequence Diagrams, this causality ordering is identical to a top-down visual ordering. In this thesis, a *message* is considered to be a Sequence Diagram fragment.

CombinedFragments are high level additions to Sequence Diagrams. They are instances of *InteractionFragment* that consists of *InteractionOperators*. *CombinedFragments* may include multiple *InteractionFragments*; which means it could consist of other *CombinedFragments*. As a consequence, *CombinedFragments* may have a hierarchical structure. This hierarchical structure is also sometimes referred to as *nested CombinedFragments*. The nesting of *CombinedFragments* may also occur between different *InteractionOperatorKinds*. There are four *InteractionOperatorKind* used in this thesis as depicted in Figure 3; *alternative*, *option*, *break* and *parallel*. Since each of the four

InteractionOperatorKind changes the flow of events in a different way, they each are designated as a fragment type.

Overall, there are five types of Sequence Diagram fragments; *message*, *alternative*, *option*, *break* and *parallel*. Each Sequence Diagram inputted into SD2PN is decomposed based on these five fragment types. The decomposition however preserves the causality of the *messages* or the hierarchical structure of the *CombinedFragments*. In the next section, these fragments are transformed into an equivalent Petri Net block.

4.1.2 Transformation

This section describes Stage 2 of the model transformation where each Sequence Diagram fragment is transformed into a corresponding Petri Net block. This requires a set of five transformation rules to be introduced; one for each type of fragment.

Before the transformation rules are presented, a destination metamodel has to be introduced. The Petri Net metamodel depicted in Figure 7of Chapter 2 corresponds to the description of standard Petri Nets presented in Section 2.2. However for the purpose of this section of the thesis, a temporary, necessary extension of Petri Nets is introduced through two new concepts;*placeholders* and *Petri Net blocks*. As such the metamodel of Figure 7is extended for the use of SD2PN, as depicted inFigure 14.

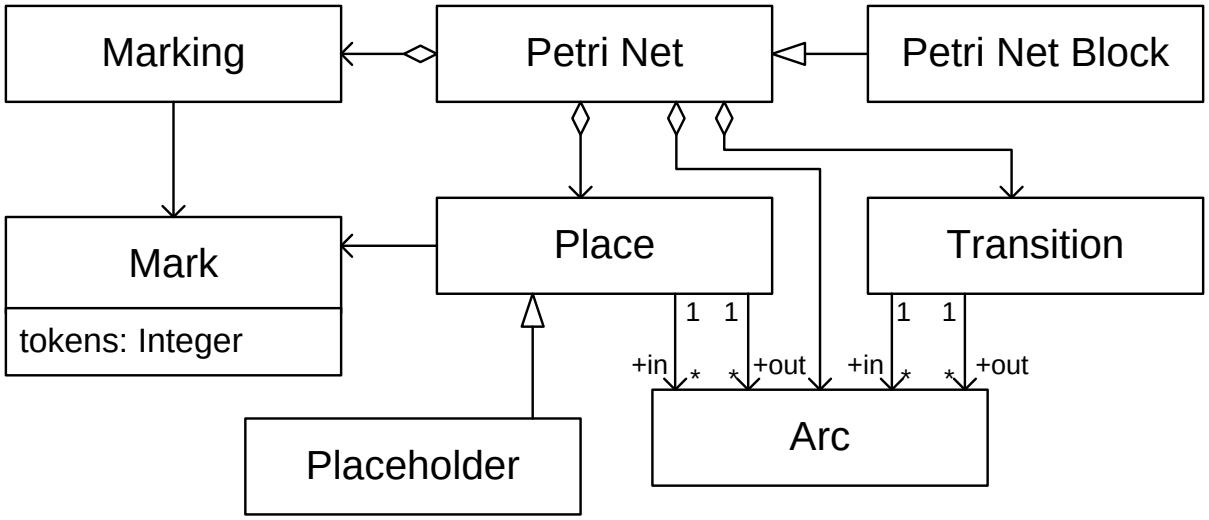

Figure 14: Extended Petri Net Metamodel for SD2PN

Definition 11:*Placeholders* are temporary nodes that mimic the structure of a *place* in Petri Nets and are depicted as dashed rectangles.

Definition 12: Petri Net blocks are blocks of Petri Nets that have unique input and output *places*, which are referred to as *precondition* and *postcondition* respectively. A more formal definition of Petri Net blocks is as follows.

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*. $In(B)$, $Out(B) \in S$ are *unique* places (*precondition* and *postcondition* respectively) such that $In(B)$ has no incoming arcs and $Out(B)$ has no outgoing arcs. They represent the start and end places in the Petri Net blocks respectively. As such, a Petri Net bock can also be textually represented as the sum of all its components. For example, the Petri Net block in Figure 15 can also be written as

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

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

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

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

$$P = \{\ \ \}$$

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

Petri Net blocks also clearly extends the definition of conventional Petri Nets, since a Petri Net block where P = ∅ is a conventional Petri Net.

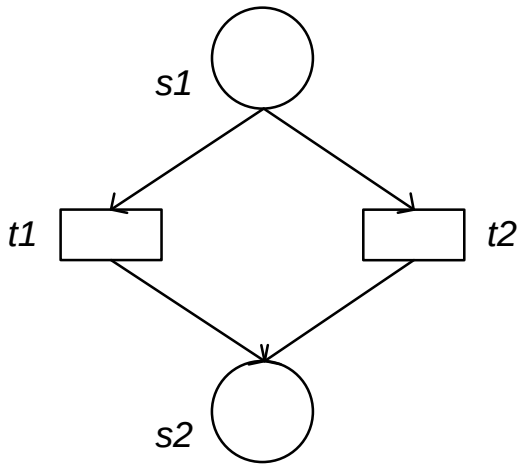

Figure 15: Example of a Petri Net block

Following the definitions of*placeholders* Petri Net blocks, the Sequence Diagram fragments can be transformed using a set of transformation rules. The description of each rule is presented in the following sections, including a graphical representation of the rule and a textual description as well as additional conditions.

4.1.2.1 Rule 1: Transforming Messages

As previously established, a *message* represents the flow of information in the system between two objects. Page 491 of [133] describes a *message* as either a call for the execution of an *operation* or depicting sending and receiving of a signal. A *message* not only signifies the type of communication or signal, it also specifies the sender and receiver. For each *message* fragment in the Sequence Diagram, an equivalent Petri Net block is generated.

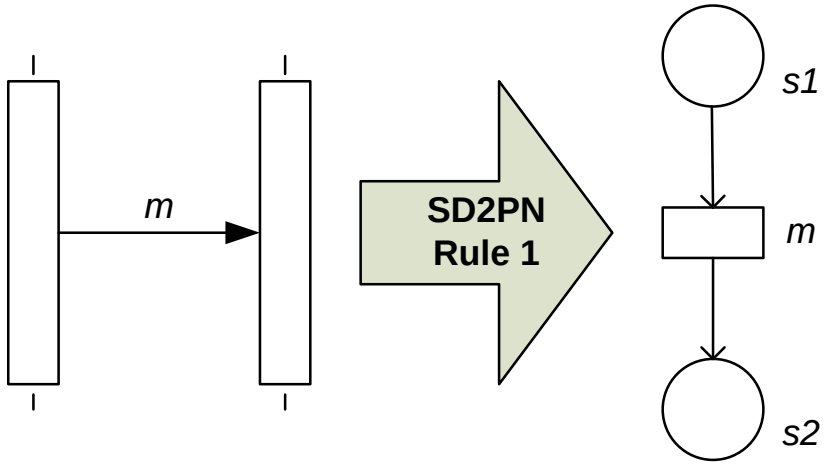

Figure 16: SD2PN Model Transformation Rule for *message* fragments

For each *message* fragment that exists in the Sequence Diagram, a Petri Net block is created. This Petri Net block consists of two *places, s1* and *s2*. These *places* signify the *precondition* and *postcondition* of the Petri Net block respectively. The *message, m* in the Sequence Diagram fragment is transformed into a *transition* in the Petri Net block and labelled with the same name. The *transition m* is connected to the *precondition* and *postcondition* via incoming and outgoing arcs respectively. This result in a Petri Net block as depicted in Figure 16, or textually as

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

There is however an additional constraint to the transformation of *messages* in SD2PN. Suppose a *messagem* in the Sequence Diagram. Suppose that *M* is the set of all *messages*in the Sequence Diagram, then for every *m* ∈ *M* where there are no *EventOccurrences* or *events* that occurs *beforem*(refer Section 2.1.1 for the description of *before* in *EventOccurrences*)[4]; the resulting Petri Net block is modified to include a *token* in its *precondition* (i.e. the *places1* in Figure 16).

4.1.2.2 Rule 2: Transforming Alternative CombinedFragments

A *CombinedFragment* with the *InteractionOperatorKind alternative* specifies a different set of events that may occur based on the conditions in the fragment [7]. The *alternative* fragment serves typically as an '*if... else...*' condition in modelling interactions or behaviour. For each *alternativeCombinedFragment* that exist in the Sequence Diagram, an equivalent Petri Net block is generated.

[4] From this point, any *messagem* ∈ *M* where there are no *EventOccurrences* or *events* that occur *beforem* could also be referred to as *first message* for brevity.

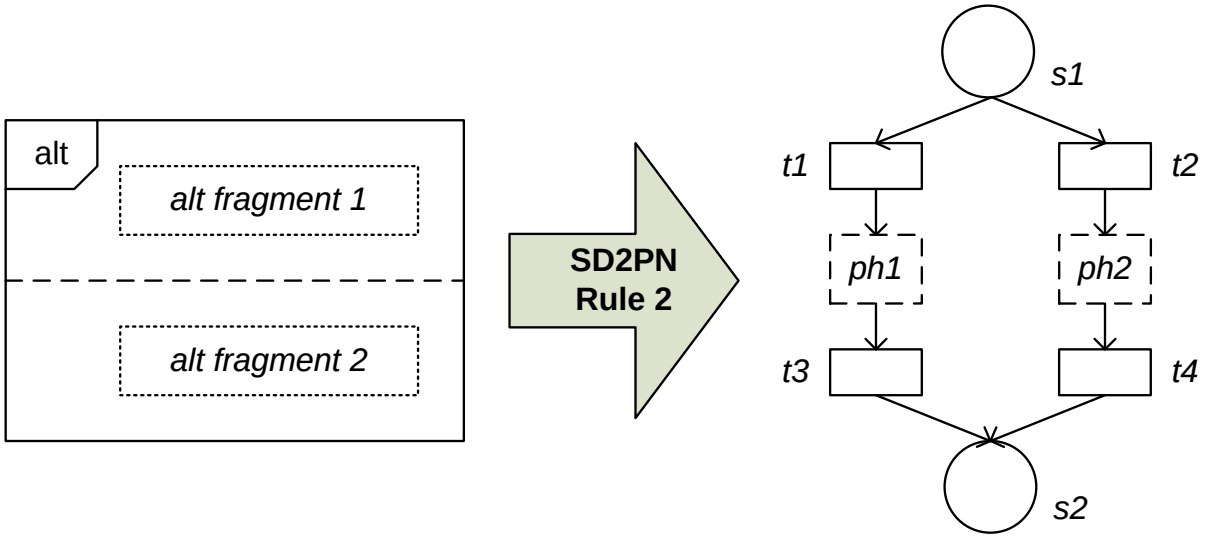

Figure 17: SD2PN Model Transformation Rule for *alternative* fragments

For each *CombinedFragment* with the *InteractionOperatorKind alternative* in the Sequence Diagram, a Petri Net block is created. The Petri Net block contains two *places*, *s1* and *s2* to model its *precondition* and *postcondition*. The Petri Net block also contains two *placeholders*, *ph1* and *ph2* as temporary *places* that will be replaced by the events in the operands *alt fragment 1* and *alt fragment 2* respectively. The behaviour of the *alternative* fragment is signified by two *transitionst1* and *t2* with incoming arcs from the *precondition*; thus only one of the two *transitions* may fire. The *transitions t1* and *t2* are connected to *ph1* and *ph2* respectively. Two more *transitions t3* and *t4* are created to denote the end of the *alternative* fragments. The *transition t3* receives and incoming arc from *ph1*,*t4* receives an incoming arc from *ph2* and both *t3* and *t4* are connected via an outgoing arc to the *postcondition*. This results in a Petri Net block as depicted in Figure 17 or written as $B = (S, T, P, F)$ where

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

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

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

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

The graphical rule and textual description above depicts a scenario of an *alternative CombinedFragment* with two operandsinside it. However, the flexibility of the rule allows for more than two operands inside the *alternative*. In such cases, each additional operandwill be transformed into two *transitions* and a *placeholder* (i.e. *t5*, *t6* and *ph3* respectively). The *placeholder ph3* is connected with the *transitions* via an incoming arc from *t5* and an outgoing arc into *t6*. The *transitions* in turn are connected to the main Petri Net block with an incoming arc from the *precondition* into *t5* and an outgoing arc from *t6* into the *postcondition*. The Petri Net block can be expanded in this way for every additional operand that exists.

Another additional condition is related to the content of the *alternative* fragments. If any of the operands inside the *alternative* fragment(including hierarchical fragments) contain the *firstmessage* of the Sequence Diagram, the Petri Net block is modified to include a *token* in the *precondition*.

4.1.2.3 Rule 3:Transforming Option CombinedFragments

The *InteractionOperatorKind option* is very similar to *alternative*. This is evident by the similarities in their construct [7]. For each *CombinedFragment* with the *InteractionOperatorKind option*, a Petri Net block is generated.

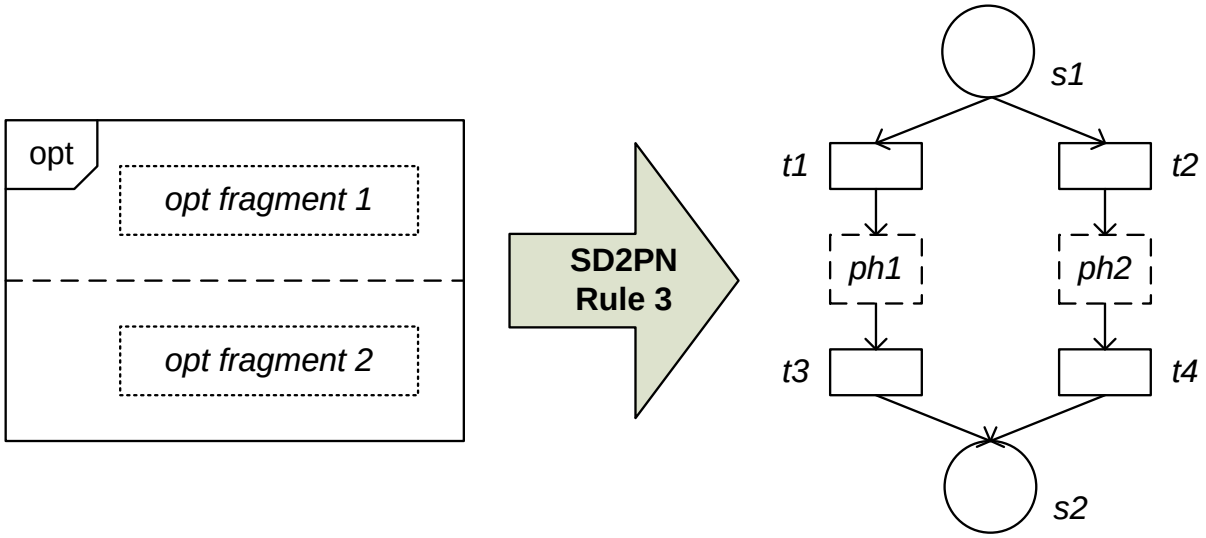

Figure 18: SD2PN Model Transformation Rule for *option* fragments

The Petri Net block that is generated for the *option* fragment is as shown in Figure 18 or $B = (S, T, P, F)$ where

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

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

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

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

The basic construct of the Petri Net block is identical to the *alternative* Petri Net block. However, the difference between the generated Petri Net block is condition the additional constraint; A *CombinedFragment* of type *option* may contain just one operand. In such cases, the Petri Net block depicted in Figure 18 is modified accordingly. Since only one operand exists, there must only be one *placeholder* in the Petri Net block. Thus, the *placeholder ph2* is replaced with a *place skip* that mimics the system where the actions inside the *option* operand are 'skipped'. The resulting Petri Net block is as shown in

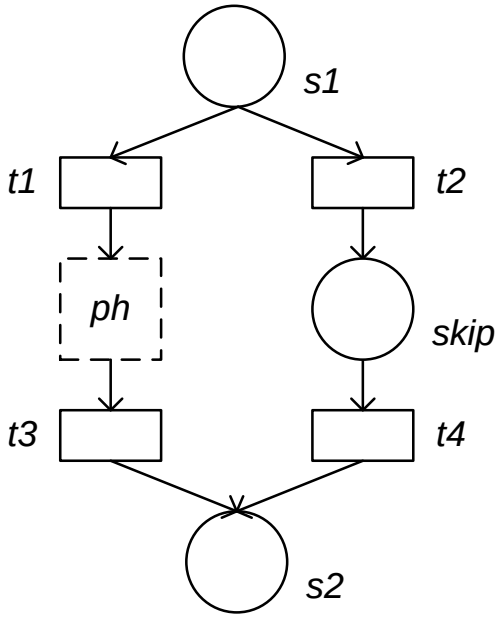

Figure 19: A Petri Net block depicting an *option* fragment with one operand

Similarly to the previous rule, if the *CombinedFragment* contains the *first message* of the Sequence Diagram (including inside nested fragments), the *precondition* of the resulting Petri Net block must contain a token.

4.1.2.4 Rule 4: Transforming Break CombinedFragments

A *breakCombinedFragment* consists of a *guard* (condition) such that when it is satisfied, the operation *breaks* (i.e. terminates) [7]. The *break* fragment is a specialization of the '*if... else...*' construct, where *if* the condition is satisfied, the system terminates. Each *break* fragment is transformed into a corresponding Petri Net block.

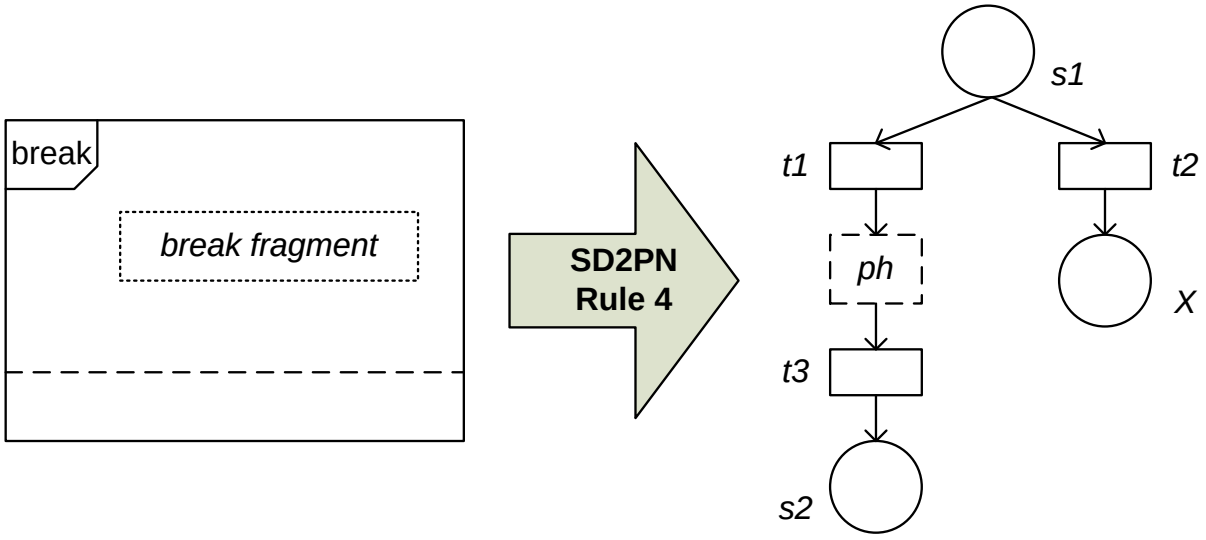

Figure 20: SD2PN Model Transformation Rule for*break* fragments

A Petri Net block is created for every *CombinedFragment* of type *break* that exists in the Sequence Diagram. This Petri Net block consists of the *precondition* and *postcondition* modelled as *places s1* and *s2*. The operand inside the *break* fragment is modelled by a *placeholder* in the Petri Net block. Similar to previous rules, two *transitions*, *t1* and *t3* are used to connect the *placeholder* to the *precondition* and *postcondition*. To illustrate termination of the system, a *place* marked by *X* is created. This is referred to as *terminal node*. The *terminal node* is connected to the *precondition* by means of a *transition t2*. However, the *terminal node* is not connected to the *postcondition*; since the system is terminated at *X*. The resulting Petri Net block can be written as $B = (S, T, P, F)$ where

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

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

$$P = \{ph\}$$

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

Unlike previous rules for *CombinedFragments*, the *break* fragment does not increase in number of operands. However, the condition for the existence of the *first message* of the Sequence Diagram remains the same. If the *break* fragment contains the *first message*, then the *precondition* of the resulting Petri Net block must contain a *token*.

4.1.2.5 Rule 5: Transforming Parallel CombinedFragments

The final rule of SD2PN is a rule that transforms every *CombinedFragment* with *InteractionOperatorKindparallel* into a Petri Net block. A *parallel*operator specifies that two or more sets of event should occur concurrently without any pre-defined set of conditions, as described in page 468 of [133]. As such, there should not be any causality or conflicting event between all the operands of the *parallel* fragment.

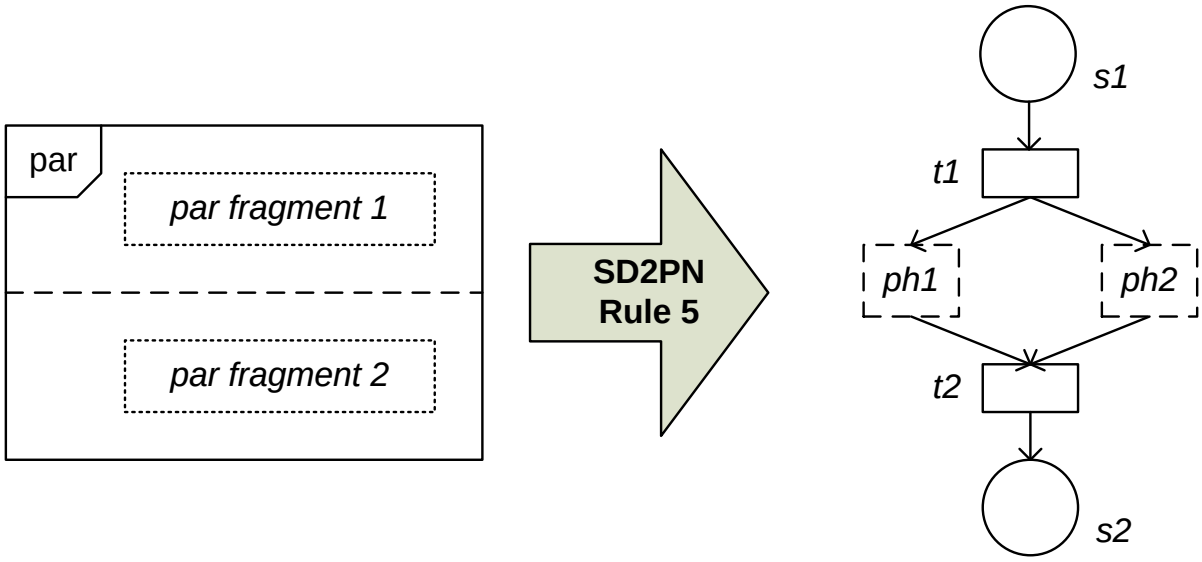

Figure 21: SD2PN Model Transformation Rule for *parallel* fragments

For each *CombinedFragment* of type *parallel* that exists in the Sequence Diagram, a Petri Net block is created. Typically, this 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. However, to model the concurrency between the operands, a single *transitiont1* is used to connect all the *placeholders* to the *precondition*. This is because the firing of *t1* will provide *tokens* to both *ph1* and *ph2*; allowing them to run in parallel. Another *transition t2* is created to connect the *placeholders* to the *postcondition*. The Petri Net block generated by SD2PN can be written as $B = (S, T, P, F)$ where

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

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

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

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

Similar to the *alternative* fragment, the *parallel* operator allows more than two operands. In the occurrence of such event, each additional operand is transformed into a single *placeholder*. This *placeholder* is connected to the main Petri Net block through and incoming arc from *t1* and an outgoing arc into *t2*. This creates concurrency between all the operands, immaterial of the number.

If the *parallel* fragment, or any of its nested fragments contains the *first message* of the Sequence Diagram, then the *place s1* (*precondition*) must be modified to include a token.

4.1.3 Composition

Following the mapping of each Sequence Diagram fragment into a corresponding Petri Net block, an integrated Petri Net that corresponds to the original Sequence Diagram needs to be produced by composing the Petri Net blocks.

Examined closely, there is a commonality between all the Petri Net blocks generated via SD2PN; each have a single input and output *place*, or as previously introduced, *precondition* and *postcondition*. This is deliberate to allow a uniform method of putting the Petri Net blocks together. There are two local functions used for this purpose: *morph* and *substitute.*

4.1.3.1 Morph

The function *morph* is used to put together causal Petri Net blocks. In formal descriptions, the symbol ⊗ is used to denote *morph* (i.e. $B_1 \otimes B_2$). The causality relationship is derived from the *GeneralOrdering* from the Sequence Diagram metamodel in Figure 3. The *morph* function is used to connect Petri Net blocks by merging the *postcondition* of a block with the *precondition* of another, enforcing a causal behaviour. The *morph* function can only be called with two Petri Net blocks at a time.

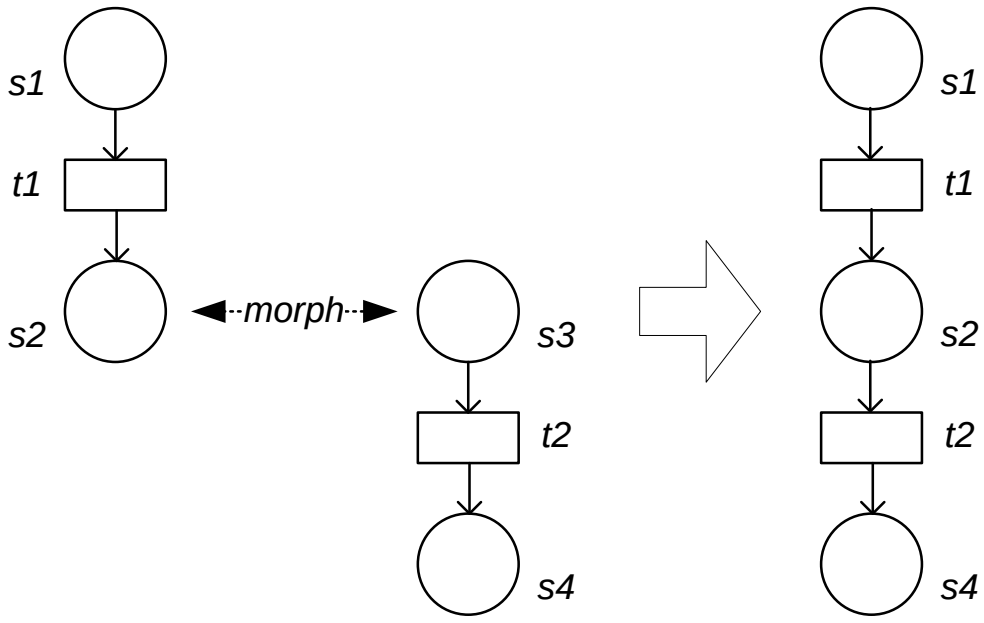

Figure 22: Example of a morph action between two Petri Net blocks

Figure 22 illustrates the concept of *morph* using an example. When the *morph* function is invoked on two Petri Net blocks, the *postcondition* of the former is merged with the *precondition* of the latter, creating an integrated Petri Net block. As can be observed in Figure 22, the *morphed* place will always take the label of the former block – disregarding the latter. A more formal description is also provided 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, denoted by $B_1 \otimes B_2$ results in a 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 (∗), notice that the arcs in B are obtained by including all the arcs in $F_1 \cup F_2$ except 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 and B_2.

As for the example in Figure 22, suppose 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 and *s2* and *s4* are *postconditions* of B_1 and B_2. Invoking the *morph* function as $B_1 \otimes B_2$ merges the *postcondition* of B_1 and the *precondition* of B_2, creating a Petri Net block such that:

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

4.1.3.2 Substitute

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

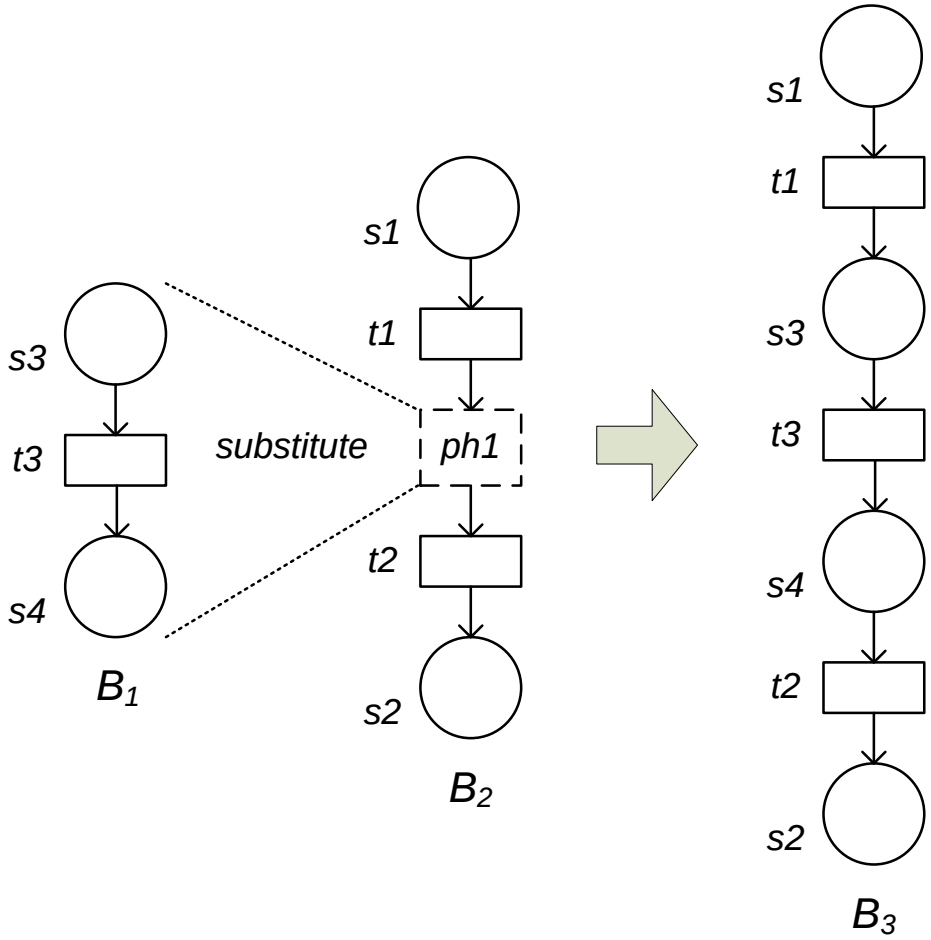

Figure 23: Example ofa *substitute* action between Petri Net blocks

Figure 23 illustrates an instance of *substitution* between two Petri Net blocks and the result of the *substitution* process. Every time the *substitute* function is invoked, a *placeholder*

is replaced by an entire Petri Net block such that the incoming arc into the *placeholder* is transferred into the *precondition* of the block, while the outgoing arc from the *placeholder* is transferred as if from the *postcondition* of the block. A more formal definition of *substitute* follows.

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. *Substituting* the Petri Net Block, B_1 into *ph1*, denoted by $B_2[B_1/ph1]$ 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

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

The equation (**) states that arcs in B can be obtained by removing all arcs to and from *ph1* and redirecting them to $In(B_1)$ and $Out(B_2)$ respectively.

Since most cases of *substitution* in SD2PN involve the need for two Petri Net blocks to be *substituted* into one Petri Net block with two *placeholders* (i.e. *alternative, parallel* and certain cases of *option*); another example is presented in Figure 24.

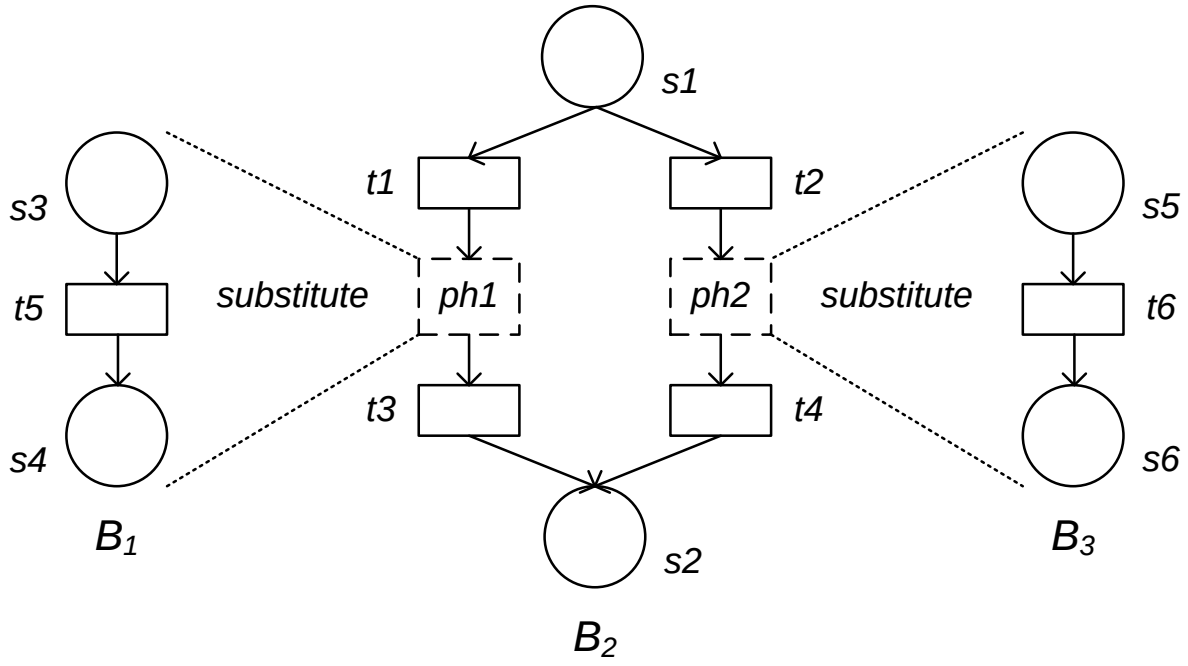

Figure 24: Example of two *substitute* actions between Petri Net blocks

As for the example in Figure 24, suppose three Petri Net blocks B_1, B_2 and B_3 such that

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

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

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

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

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

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

$$P = \{\quad\}$$

$$F = \begin{Bmatrix}(s1, t1), (s1, t2), (t2, s3), (s3, t5), (t5, s4), (t2, s5),\\ (s5, t6), (t6, s6), (s4, t3), (s6, t4), (t3, s2), (t4, s2)\end{Bmatrix}$$

In cases where the *substitution* is between three[5] Petri Net blocks such that $B_1$and B_3 substituted into B_2 as presented in Figure 24; the order by which the *substitution* takes place is arbitrary. The *substitution* of $B_2[B_1/ph1]$ followed by $B_2[B_3/ph2]$ generates the same result as the *substitution* of $B_2[B_3/ph2]$ followed by $B_2[B_1/ph1]$. As a result;

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

4.2 SD2PN Generates Free Choice Petri Nets

Following the definition of the SD2PN model transformation, an interesting and vital observation was made: SD2PN generates only Free Choice Petri Nets. Free Choice Petri Nets is a well studied subclass of Petri Net where *conflicts* and *concurrencies* may occur but not simultaneously.

Recalling the definition of Free Choice Petri Nets from Section 2.2.2, Baccelli [38] defines Free Choice Petri Nets, as whenever two transitions in the net share an input place, they must not have any other input places. This can also be written as for every *s*, when $|s^\circ| > 1$, for every $t \in s^\circ, |{}^\circ t| = 1$.

[5] This is also true for cases where more than two Petri Net blocks are *substituted* into one Petri Net block; however it is not explained since there are no Petri Net blocks with more than two *placeholders* in the SD2PN transformation rules.

This section aims to prove the claim that SD2PN generates only Free Choice Petri Nets. To begin with, an extension of Free Choice Petri Nets to Petri Net blocks is defined. A Free Choice Petri Net Block is a Petri Net Block, B = (S, T, P, F) such that for each s ∈ S ∪ P, if $|s^{\circ}| > 1$, then every $t \in s^{\circ}, |{}^{\circ}t| = 1$.Lemma 1 is derived directly from this definition.

Lemma 1: A Free Choice Petri Net block with no *placeholders* is a Free Choice Petri Net.

Proof: Trivial.

Lemma 2: Every individual transformation rules of SD2PN generates a Free Choice Petri Net block.

Proof: Rules 1 and 5 has no instance of $|s^{\circ}| > 1$, where else in Rules 2,3 and 5, in the instance where $|s^{\circ}| > 1$, every $t \in s^{\circ}, |{}^{\circ}t| = 1$. Therefore, every individual transformation rules of SD2PN generates Free Choice Petri Net blocks.

Lemma 3: The set of Free Choice Petri Net blocks are closed under *morph* and *substitution*, i.e. if B_1 and B_2 are Free Choice Petri Net Blocks, then $B_1 \otimes B_2$ and $B_2[B_1 / p]$ where p is a *placeholder* in B_2, are also Free Choice Petri Net Blocks.

Proof: To show that $B_1 \otimes B_2 = (S, T, P, F)$ is a Free Choice Petri Net Block, suppose $s \in S \cup P$ such that $|s^{\circ}| > 1$, then s is either a *place* or a *placeholder* in B_1 or B_2, since $s \neq Out(B_1)$ because $|Out(B_1)^{\circ}| = 0$. In either case, since both B_1 and B_2 are Free Choice Petri Net Blocks, then $B_1 \otimes B_2$ is also a Free Choice Petri Net Block since $B_1 \otimes B_2$ does not create a new scenario such that $|s^{\circ}| > 1$.

To show that $B_2[B_1/p] = (S, T, P, F)$ is a Free Choice Petri Net, we suppose that p is a *placeholder* in B_2. The process of *substitution* replaces all arcs into p and redirects them into

In($B_{2)}$ and redirects *Out*(B_2) into the output of *p*. This does not incur any new situation such that $|s^{\circ}| > 1$, because the redirection of arcs is a direct mapping from one node to another. Therefore $B_2[B_1/p]$ is a Free Choice Petri Net Block.

Theorem 1: Every Petri Net generated via SD2PN is a Free Choice Petri Net.

Proof: As previously described, there are three stages in the SD2PN model transformation. The first stage of the model transformation decomposes the Sequence Diagrams into fragments. In stage 2, each fragment is transformed into Petri Net blocks.From Lemma 2, it is established that the Petri Net blocks are all Free Choice. Stage 3 puts together the Free Choice Petri Net blocks using *morph* and *substitution*. By Lemma 3, the set of Free Choice Petri Net blocks are closed under *morph* and *substitute*.Stage 3 ends when all the Petri Net blocks are connected and there are no more *placeholders*. By Lemma 1, a Free Choice Petri Net Block with no *placeholders* is a Free Choice Petri Net. Thus, SD2PN only generated Free Choice Petri Nets.

4.3 SD2PN Preserves Semantics

As discussed in Chapter 3, preservation of semantics is vital in Multi Paradigm Modelling. In this case, it is essential for the resulting Petri Net to retain the same behavioural properties as the original Sequence Diagram. In this section, the term *correctness* refers to the preservation of semantics between the source model and the destination model.

In order to show that each Sequence Diagram fragment type described in Section 4.1.1 and the corresponding Petri Net block presented in 4.1.2 consist of the same behaviour, the

semantics of both the Sequence Diagram fragment and the Petri Net block is compared. As such, a common semantic domain is required.

Labelled Event Structures (LES) is chosen as the common semantics domain since both Sequence Diagrams and Petri Nets can be depicted as LES using techniques from [40] and [41] respectively Furthermore LES offers a similar approach to modelling when compared to both Sequence Diagrams and Petri Nets such that all three languages focus on behaviours and flow of events. LES also offers the ability to model causality, conflicts and concurrencies; the three types of relationships offered in Sequence Diagrams.

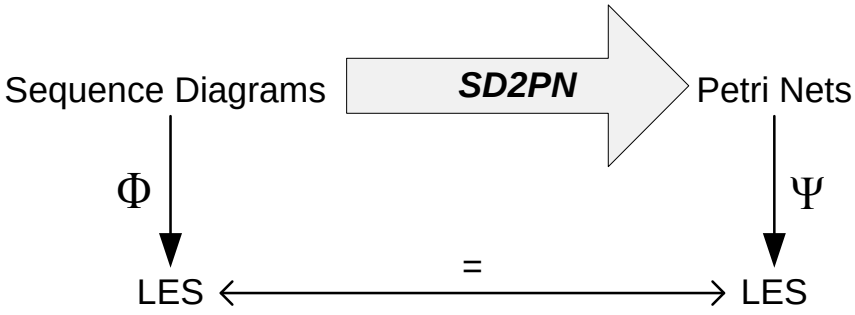

Figure 25: Using LES as a common semantics domain to prove correctness

Figure 25depicts the outline of the approach in which Φ is a semantic map introduced by Küster-Filipe [40] and Ψ is a semantic map introduced by McMillan [41] used in unfolding of Petri Nets. Using these semantic maps, Sequence Diagrams fragments and Petri Net blocks are mapped into LES; and the comparison between the LES is used as the proof that the SD2PN model transformation preserves the semantics of the Sequence Diagrams in the resulting Petri Nets. Preliminary information about LES and how Sequence Diagrams and Petri Nets map into LES can be recalled from Section 2.3.

In order to prove that SD2PN preserves semantics, both the LES generated from the Sequence Diagram and Petri Net has to be equal. This is shown using the following Lemmas.

Lemma 4: Every Sequence Diagram fragments and its corresponding Petri Net block created by SD2PN generate the same LES.

Proof: As previously established, there are five types of Sequence Diagram fragments; *message*, *alternative*, *option*, *break* and *parallel.* Since the *alternative* and *option* fragments are semantically equivalent, they will be grouped as one fragment for the purpose of this proof.

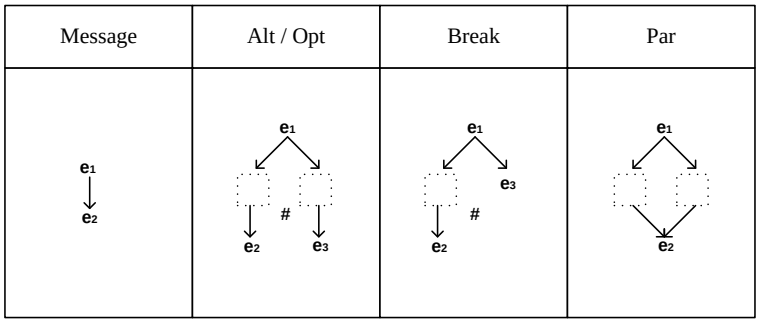

Figure 26: LES obtained from Sequence Diagram fragments and each corresponding Petri Net blocks

Each type of Sequence Diagram fragment is translated into LES based on the semantic mapping Φ while the corresponding Petri Net block is unfolded into LES using the semantic mapping Ψ. This results in a Labelled Event Structure for each type of fragment, as depicted in Figure 26. Both the semantic maps are explained in Section 2.3. Due to the tedious nature of the individual translation into LES, an example using the *parallel* Sequence Diagram fragment is presented below while the remainder of the proof is presented in Appendix B.

In Sequence Diagrams, a *parallel* fragment has an initial location l_1. This location signifies the beginning of the fragment. Inside the fragment, there are 2 scopes; par(2)#1 and par(2)#2 as described in Section 2.3. These scopes represent the parallel events that occur inside the fragment. After the execution of these events, a location l_2 signifies the end of the fragment. Since both l_1 and l_2 has an *alt_loc* of 1, there is only 1 event to represent each these

locations, e_1 and e_2 such that e_1 forks into the 2 scopes of events and merge into e_2. This creates an LES as shown in Figure 26.

In the corresponding block of Petri Net, it starts with a place s_1 and ends with a place s_2. They can be represented as events e_1 and e_2 respectively with e_1 forking out into the *placeholders* and merging at e_2. This is exactly the same as the representation in Figure 26, thus showing that the transformation preserves the behaviour of the original Sequence Diagram.

Lemma 5: In ordering of the Sequence Diagram fragments, every two fragments and the corresponding block of Petri Net that it maps into consist of the same semantics.

Proof: For proof of Lemma 5, every possible permutation of events that may occur in the transformation is considered. There are two possible connectors between Petri Net blocks, as previously established; *morph* and *substitute*. Figure 27shows every possible permutation of events including all *morph* and *substitution* cases. Cases that allow both techniques are labeled as*mor* representing *morph* and *sub* representing *substitution*.

Carrying from the proof of Lemma 4, for every two Sequence Diagram fragment, the translation into LES is performed using the semantic map Φ. This is followed by the unfolding of the corresponding Petri Net blocks into LES using Ψ. By comparison, the LES generated through both semantic maps are identical, as shown in Figure 27. By mapping every permutation of Sequence Diagrams fragment and the corresponding Petri Nets into LES as presented in Figure 27[6]; it is proven that every two Sequence Diagram fragments and the corresponding block of Petri Net that it maps into consist of the same semantics.

[6] The translation of all sixteen permutations of Sequence Diagram fragments, and the unfolding of their corresponding Petri Nets into LES are performed similarly to the mapping presented in Appendix B and as such, it is omitted due to word restrictions.

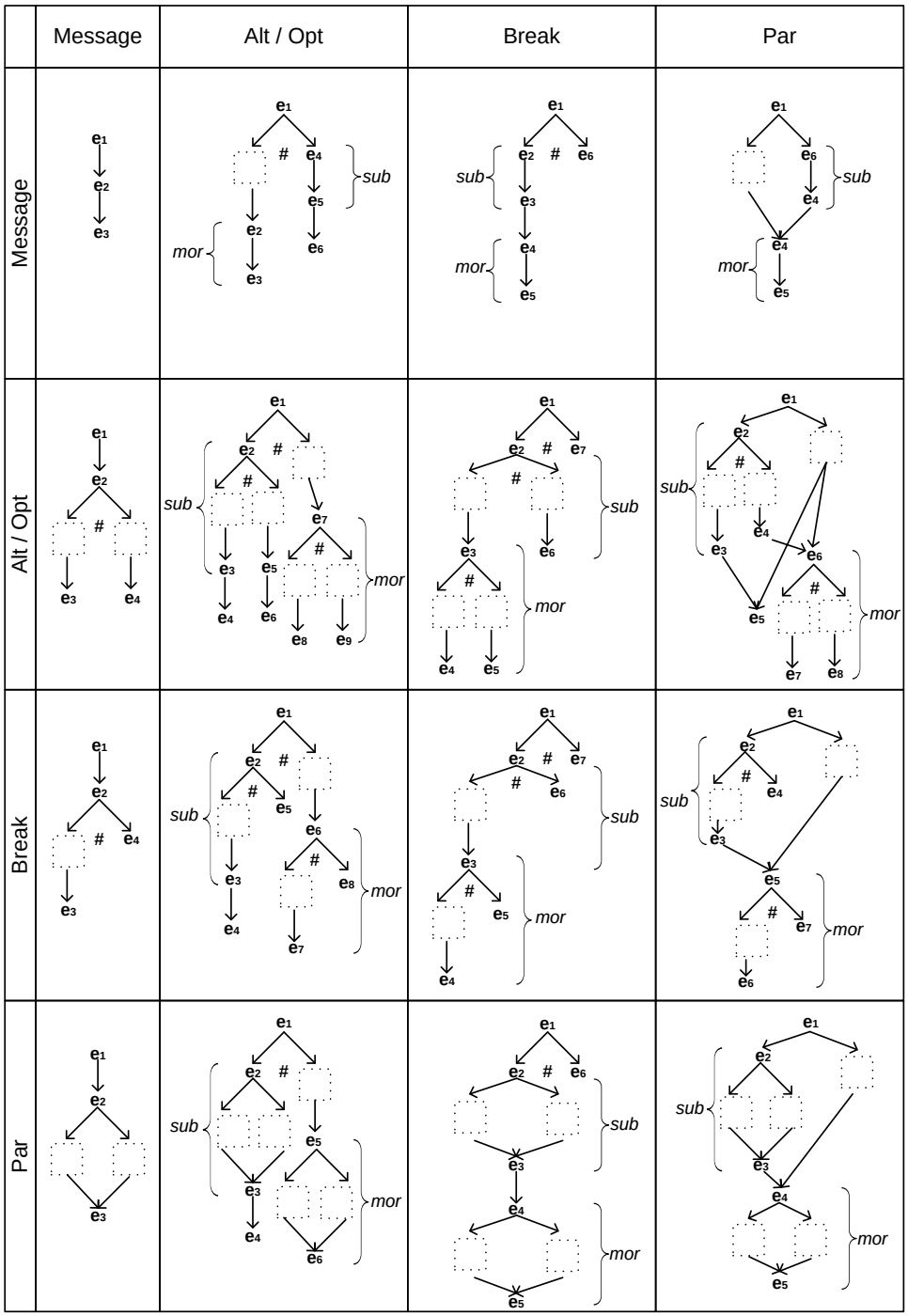

Figure 27: LES generated by every two Sequence Diagram fragments and their corresponding Petri Net blocks

CHAPTER 5

SEQUENCE DIAGRAM ANALYSIS VIA SD2PN

This chapter presents an approach for the analysis of Sequence Diagram based on the Multi Paradigm Modelling platform using SD2PN.

5.1 Importance of Analysis in Sequence Diagram

Sequence Diagrams are used to model dynamic aspects of a system. The dynamic aspects of a system include flow of events in various situations, interaction between various components of the system, as well as user interaction with the system. An accurately modelled Sequence Diagram is vital, and can be used not only in conveying information between stakeholders, but even in *forward* and *reverse* engineering of executable systems [134].

In this respect, forward engineering refers to code generation from Sequence Diagrams where the accuracy of the Sequence Diagram determines the build quality of the

system. Code generation based on Sequence Diagrams that contain errors result in the creation of flawed systems that may require a thorough code examination or worse, remodelling. Performing model level analysis on the Sequence Diagrams before the codes are generated could potentially save countless man-hours and resources from being wasted in developing a flawed system. For example, a Sequence Diagram that is free from deadlock ensures a system without deadlock. However, model analysis is not meant to replace the testing phase in the system development cycle. The responsibility of performing analysis on the Sequence Diagram should be on the system designers. As such, any errors in the design could be rectified instantaneously ensuring only the highest quality, error-free Sequence Diagrams are used in the code generation. This therefore reduces the probability of a flawed system.

Reverse engineering on the other hand refer to the creation of Sequence Diagrams from codes or complete systems. Reverse engineering is a tricky process that could lead to complex Sequence Diagrams which contain too much information [134]. This leads to redundancy in the Sequence Diagrams, which in turn results in inaccurate representation of the system. Performing model analysis on the Sequence Diagram could pinpoint the errors and highlight the necessary changes to make the Sequence Diagram more accurate in its depiction of the system.

5.2 Implementing SD2PN for Analysis of Sequence Diagrams

SD2PN, in Chapter 4 is introduced as a MDD model transformation that transforms Sequence Diagrams into Free Choice Petri Nets. Based on the need for model analysis to be performed

on Sequence Diagrams and the suitability of Petri Nets for performing such analysis; SD2PN provides a framework that allows the analytical capabilities of Petri Nets to be utilised on Sequence Diagrams.

SD2PN promotes model interoperability [124, 125] between Sequence Diagrams and Petri Nets, through Multi Paradigm Modelling supporting a seamless transition between these heterogeneous models. This is depicted in Figure 28 where a system designer models a system in Sequence Diagram using UML tools, then uses SD2PN to transform the models into Petri Nets. This allows complex analysis to be performed on the system using Petri Net tools. Finally, the system designer receives the analysis result in the form of a feedback. Using an automated model transformation framework as shown in Figure 28 allows the system designer to bridge the gap between the design and analysis phases of system design without extensive knowledge of Petri Nets.

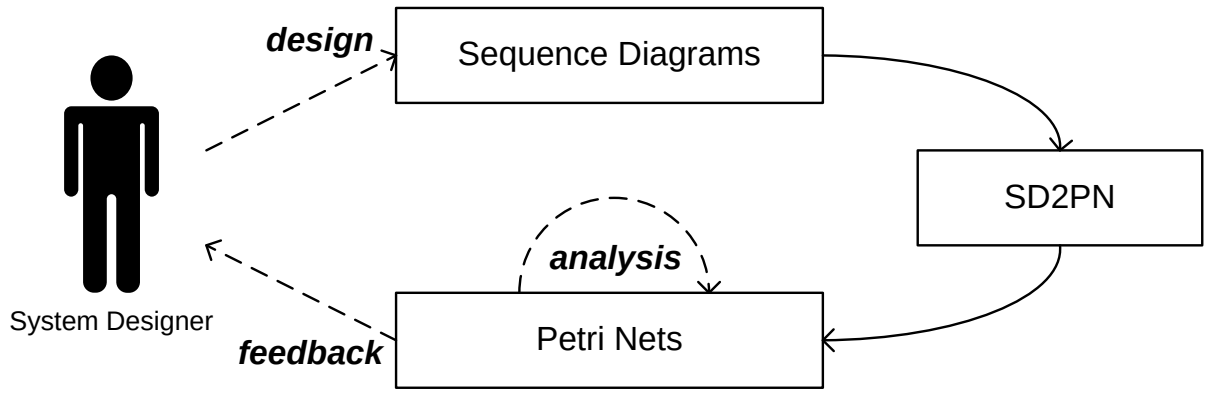

Figure 28: SD2PN Framework for Analysis

There are various types of analysis that could be performed on Petri Nets; some of which are introduced in Section 2.2.3. By transforming Sequence Diagrams into Petri Nets, the analysis capabilities of Petri Nets are essentially applied on Sequence Diagrams. This

section describes how the analytical capabilities of Petri Nets can be applied in Sequence Diagrams.

In Petri Nets, liveness translates to a complete absence of deadlock. This is also applicable to Sequence Diagrams. There are approaches that allows liveness analysis to be performed on Sequence Diagrams such as [135] where an *automata* based formal semantics is assigned to the Sequence Diagram and also [136] where OCL constraints are integrated with the Sequence Diagrams in order to conduct liveness analysis. This research on the other hand, takes advantage of the mathematical foundation behind Petri Nets to perform liveness analysis in Sequence Diagrams via SD2PN. Based on SD2PN transformation rules, each message in the Sequence Diagram is represented by a transition in the corresponding Petri Net. The liveness criteria in Petri Net reflect that each and every transition in the Petri Net must be enabled following a firing sequence that begins from the initial marking. As such, a live Sequence Diagram means every message in the Sequence Diagram can be reached, thus ensuring a complete absence of deadlock in the Sequence Diagram.

The liveness analysis is essential in modelling a system.Sequence Diagrams model the behaviour and interactions within a system through messages, thus it is vital for each message to execute correctly. This could be achieved through liveness analysis. Proving the absence of deadlock in the system also prevents unpredictable behaviours in the system such as unexpected termination of a procedure.

The reachability analysis in Petri Nets on the other hand, calculates if a particular marking is reachable from any markings in the Petri Net. This allows the analyst to ascertain what (sequence of) actions, if any, may lead to a particular state in the Petri Net. This translates well for system design using Sequence Diagrams.

In system design, this analysis could be beneficial from various points of view. For example in security terms, a reachability analysis could be performed to predict the possibility of an attack scenario. Using the same example, a reachability analysis could also be used to calculate precisely the sequence of actions that may result in the particular scenario, thus providing necessary information to the system designer to neutralize the threat. Not only that, a designer could also try to locate any specific flaws in the system by pin-pointing to a state and performing an analysis to determine what states are reachable, and what states are not.

An added advantage of performing analysis of Sequence Diagrams via SD2PN is that every Petri Net generated using SD2PN is a Free Choice Petri Net, as proven in Theorem 1 of Chapter 4. This allows analysis to be performed with a lower complexity to general Petri Nets – thus allowing the analysis results to be obtained much faster. Details about the benefits of analysis in Free Choice Petri Nets are presented in Chapter 2.

5.3 Automated Analysis via SD2PN Transformer

The MDD model transformation SD2PN provides a framework for Sequence Diagrams to be transformed into Petri Nets. This section discusses how this model transformation is automated as a tool called SD2PN Transformer[7], and how it could be used to perform model analysis. This section presents a brief introduction of the tool while the complete code for the tool is available in Appendix C.

SD2PN Transformer is a Java tool built on the platform of SiTra [89, 111] utilising the model transformation algorithm of SD2PN. This tool accepts Sequence Diagrams as inputs

[7] The coding for SD2PN Transformer was done by Behrang Sabeghi Saroui under my supervision, based on my algorithm and tool architecture.

and produces corresponding Petri Nets as outputs. At present, the tool accepts Sequence Diagrams in the form of XML Metadata Interchange (XMI) [137] and presents the Petri Nets in the form of XML [138] as depicted in the outline of the tool in Figure 29. However, there are plans to fully automate the tool by allowing integration between SD2PN Transformer, a UML tool and a Petri Net tool.

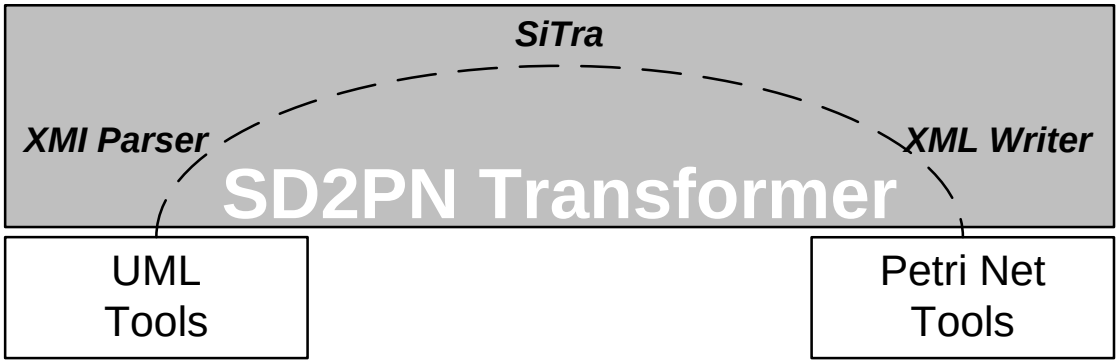

Figure 29: An outline of SD2PN Transformer

The execution of each transformation from Sequence Diagram to Petri Net in SD2PN Transformer goes through five stages; (i) generating XMI for Sequence Diagrams, (ii) parsing the XMI data into Java objects, (iii) applying the model transformation using SiTra, (iv) generating XML for the resulting Petri Net and finally (v) performing analysis on the Petri Net model using a Petri Net tool. These stages are further explained in the upcoming sections.

5.3.1 Generating XMI for Sequence Diagrams

XMI or XML Metadata Interchange [137] is a standard created by Object Management Group to allow the interchange of metadata information. XMI is commonly used to express UML models and as such, widely accepted as a form of output in UML tools. UML tools such as ArgoUML [139] and Poseidon [140] for example allow UML models designed within the tool

to be exported as XMI files. An example of a small snippet of XMI that represents a Sequence Diagram created using Poseidon is shown in Table 5.

Table 5: Snippet of XMI for a Sequence Diagram

```
<UML:GraphElement.semanticModel>
<UML:SimpleSemanticModelElement xmi.id = 'Im1ec36c75m11b56ca071dmm7b4b'
        presentation = '' typeInfo = 'SequenceDiagram'/>
</UML:GraphElement.semanticModel>
<UML:GraphElement.contained>
<UML:GraphNode xmi.id = 'Im1ec36c75m11b56ca071dmm7b2d' isVisible = 'true'>
<UML:GraphElement.position>
<XMI.field>300.0</XMI.field>
<XMI.field>20.0</XMI.field>
</UML:GraphElement.position>
<UML:GraphNode.size>
<XMI.field>100.0</XMI.field>
<XMI.field>192.0</XMI.field>
</UML:GraphNode.size>
<UML:DiagramElement.property>
<UML:Property xmi.id = 'Im1ec36c75m11b56ca071dmm7b1f' key = 'fill' value = '#ffffff'/>
<UML:Property xmi.id = 'Im1ec36c75m11b56ca071dmm7b1e' key = 'fill-opacity'
            value = '1.0'/>
<UML:Property xmi.id = 'Im1ec36c75m11b56ca071dmm7b1d' key = 'font-color'
            value = '#000000'/>
<UML:Property xmi.id = 'Im1ec36c75m11b56ca071dmm7b1c' key = 'font-family'
            value = 'SansSerif'/>
<UML:Property xmi.id = 'Im1ec36c75m11b56ca071dmm7b1b' key = 'font-size'
            value = '11'/>
<UML:Property xmi.id = 'Im1ec36c75m11b56ca071dmm7b1a' key = 'stroke' value = '#000000'/>
</UML:DiagramElement.property>
```

The snippet of XMI shown in Table 5 represents only a minute fragment of code that forms the entire Sequence Diagram. Furthermore, the code is also incomprehensible to most users and decoding the XMI to obtain information regarding the Sequence Diagram is a tedious process. However, this could be done using XMI parsers.

5.3.2 Parsing XMI Data into Java Objects

Parsing is a process of syntactical analysis and interpretation of a structured text. In this case, parsing is required to interpret the XMI code generated from UML tools into Java objects that could be manipulated by SiTra. The parsing is performed using SDMetrics [141], a tool that allows XMI input from various different UML tools using its custom XMI import function. This XMI data is then analysed and condensed into a text file consisting all the information regarding the Sequence Diagram. The information in the text file is then extracted using a custom Java program that is created to create Java objects that corresponds to the original Sequence Diagram.

5.3.3 Model Transformation via SiTra

Following the parsing of the XMI data into Java objects, the actual model transformation process takes place. SiTra provides a platform for this model transformation to take place. Recalling the introduction of SiTra in Chapter 2, a typical model transformation requires a source metamodel, a destination metamodel and a set of transformation rules to be written in Java. Snippets of code from the Sequence Diagram metamodel, Petri Net metamodel and SD2PN model transformation rule are presented in

Table 6: Snippet of code for the Sequence Diagram metamodel

```
package sequencediagram;
public class Message {
   private String id;
   private String label;
   private String context;
   private EventOccurrence sendEvent;
   private EventOccurrence receiveEvent;
   public Message(String id, String label, String context, EventOccurrence sendEvent, EventOccurrence receiveEvent) {
      this.id = id;
      this.label = label;
      this.context = context;
      this.sendEvent = sendEvent;
      this.receiveEvent = receiveEvent;
   }
   public final String getID() {
      return id;
   }
   public final String getLabel() {
      return label;
   }
   public final String getContext() {
      return context;
   }
   public final EventOccurrence getSendEvent(){
      return sendEvent;
   }
   public final EventOccurrence getReceiveEvent(){
      return receiveEvent;
   }
   public final void setSendEvent(EventOccurrence eo){
      sendEvent = eo;
   }
   public final void setReceiveEvent(EventOccurrence eo){
      receiveEvent = eo;
   }
}
```

Table 7: Snippet of code for the Petri Net metamodel

```
package petrinet;
public class Transition {
   private String name;
   public Transition(String name) {
      this.name = name;
   }
   public void setName(String newName) {
      name = newName;
   }
   public String getName() {
      return name;
   }
}
```

Table 8: Snippet of code for the SD2PN model transformation rule

```
package sitra;
import petrinet.Arc;
import petrinet.Mark;
import petrinet.PetriNet;
import petrinet.Place;
import petrinet.PlaceHolder;
import petrinet.Transition;
import sequencediagram.CombinedFragments;
import sequencediagram.InteractionOperatorKind;
public class Rule3 implements Rule {
   public Rule3() {
   }
   public boolean check(Object source) {
      return (source instanceof CombinedFragments) && (((CombinedFragments) source).getOperator() == InteractionOperatorKind.OPT);
   }
   public PetriNet build(Object source, Transformer t) {
      PetriNet pn = new PetriNet(((CombinedFragments) source).getContext(), ((CombinedFragments) source).getID(), "OPT");
      Place p1 = new Place(new Mark(0));
      Place p2 = new Place(new Mark(0));
      for (int j = 0; j < ((CombinedFragments) source).getNumberOfFragments(); j++) {
         Transition trans = new Transition("OPT" + j);
         pn.addArc(new Arc(p1, trans, Arc.PLACE_TO_TRANSITION));
         PlaceHolder ph1 = new PlaceHolder(((CombinedFragments) source).getContext(), ((CombinedFragments) source).getFragments()[j], "PH" + j);
         pn.addArc(new Arc(ph1, trans, Arc.TRANSITION_TO_PLACEHOLDER));
         Transition transEnd = new Transition("END-OPT" + j);
         pn.addArc(new Arc(ph1, transEnd, Arc.PLACEHOLDER_TO_TRANSITION));
         pn.addArc(new Arc(p2, transEnd, Arc.TRANSITION_TO_PLACE));
      }
      return pn;
   }
   public void setProperties(Object target, Object source, Transformer t) {
   }
}
```

The execution of the model transformation results in Petri Net Java objects that is based on the Petri Net metamodel in SiTra. However, for these Petri Net Java objects to be recognized by any Petri Net tools, it has to be written into an XML form recognized by the particular tool.

5.3.4 Generating XML for Resulting Petri Net

There are numerous Petri Net tools available in the market, among them CPNTools [27] and PIPE [28]. However, each Petri Net tool uses a different flavour of XML to represent the Petri Nets. This results in the need to create an XML writer for each type of Petri Net tool. The process of creating an XML writer to conform to a certain Petri Net tool involves tedious analysis of the tool's output to recognize the patterns in the XML. This process can be exhaustive and redundant.

At present, a solution for this problem has not been obtained. The XML writer in SD2PN Transformer is configured to output the Petri Net in a textual format, not conforming to any Petri Net tools. However in the future, this problem could be eliminated by integrating the different toolsets of SD2PN Transformer and Petri Net tool, thus negating the need for an XML writer.

5.3.5 Utilising Existing Petri Net Tools for Analysis

The final stage of using SD2PN Transformer for analysis of Sequence Diagrams involves utilizing existing Petri Net tools such as CPNTools and PIPE to perform analysis on the resulting Petri Net. Analysis methods described in Section 2.2.3 such as liveness, boundedness, reachability, reversibility and persistence as well as various other analysis methods could be performed using the Petri Net tools without the need for extensive Petri Net knowledge by the system designer.

Presently, some knowledge of the Petri Net tool of choice is required from the system designer such as familiarity with the analysis functions offered by the tool. However as previously mentioned, future plans to integrate various toolsets with the SD2PN Transformer

could theoretically allow system designers with little to no knowledge of any Petri Net tool to perform analysis on their Sequence Diagram models.

5.4 Example

It has been previously established that Petri Nets are highly suitable for analysis, and that SD2PN creates a platform that allows system designers to take advantage of Petri Nets analytical prowess by transforming Sequence Diagrams into Petri Nets with the help of SD2PN Transformer. In this section, the usability of this concept is illustrated with the help of an example. This example is not only used to draw attention to the capabilities of SD2PN, but also to highlight a limitation of SD2PN. The following sections introduces the scenario of the example as well as the protocol behind it, followed by Sequence Diagram of the scenario, the corresponding Petri Net generated via SD2PN and the analysis of said Petri Net. The limitation of SD2PN as highlighted by the example is considered in the discussion section.

5.4.1 Introduction of the Scenario

This section provides a brief preliminary look at the scenario in this example. The example features the behaviour of a Personal Area Network (PAN). A typical example of a PAN consists of a number of stations and a router. However, an unseen element in the PAN is the medium between the stations and the router. In order to send a packet to the router, the stations in the PAN would have to compete to gain access to the medium. Thus, the more stations there are in a PAN, the larger the maximum waiting time for a single station to gain access to the medium. To deal with this, various protocols have been introduced within the

IEEE 802.11 standard. However in this example, a specific protocol within the IEEE 802.11 standard is modelled using Sequence Diagrams and transformed into a Petri Net for analysis.

Figure 30 presents a simplified PAN that has two stations and a Wireless Router that serves as an access point to the Internet. In the router, the basic IEEE 802.11 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol is used [142]. This scenario is modelled using a Sequence Diagram, where a single station attempts to gain access to the medium in order to send a packet to the router, based on the CSMA/CA protocol. Before that, the next section briefly introduces the CSMA/CA protocol.

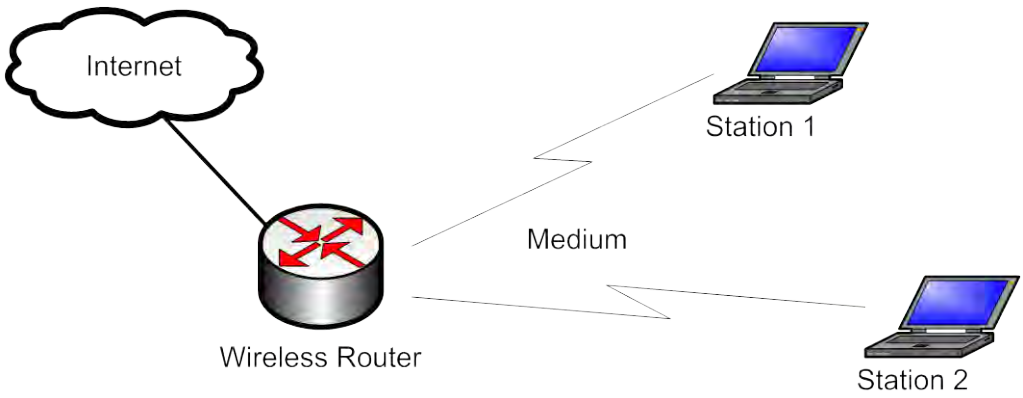

Figure 30: Overview of the Personal Area Network (PAN)

5.4.2 Protocol Description

As previously mentioned, the router in this example uses a basic IEEE 802.11 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol. CSMA/CD assigns different *waiting time* to packets in order to manage the access of the stations to the medium. There are three different waiting times for various types of packets. The shortest waiting time for medium access is called *Short inter-frame spacing* (SIFS) which is used for short control

messages or polling responses. The waiting time for time-bounded service such as a poll from the access point is considered *PCF inter-frame spacing* (PIFS) and the longest waiting time and lowest priority, *DCF inter-frame spacing* (DIFS) is used for asynchronous data services. There is a mechanism called *contention window* (CW), which is introduced in order to facilitate collision *avoidance*. The contention window makes use of an integer value that starts with CW_{min} = 7 and doubles every time a collision occurs. Every time a station tries to gain access to the medium, a random number is generated between 0 and CW and is added to the waiting time. This ensures that the stations do not send their packets at the same time. CW is doubled for every collision that occurs to accommodate a larger number of stations vying for the access of the medium. Readers are referred to [142] for more information.

Several assumptions were made in this example for the sake of clarity and to provide a better understanding of the tool. Firstly, the waiting time for all packets is constant and all packets are categorized as DIFS. Secondly, the CW is constant and does not increase, and since there are only two stations, the CW would be minimum, i.e. CW_{min} = 7. Thirdly, the packets are dropped after the unsuccessful tries from the station and each station sends only one packet. These assumptions do not invalidate the results of the analysis by any means; they onlylimit the scope of this example.

5.4.3 Sequence Diagram Representation of the Scenario

This section presents a Sequence Diagram of the scenario described in the previous sections. The Sequence Diagram in Figure 31 gives an overview of how a station sends a packet to the medium in the IEEE 802.11 protocol. The medium access control (MAC) layer of the station receives a packet from an application and registers it. It then idles before checking the status of the medium. If the medium is free the station is able to send the packet across to the

medium. However, if the medium is busy the station has to wait until the medium is free before idling again. The MAC then checks the status of the medium again before either sending the packet across or waiting again. Each of the events in this scenario has multiple sub-events that occur in the background. The diagram is however simplified for the sake of clarity.

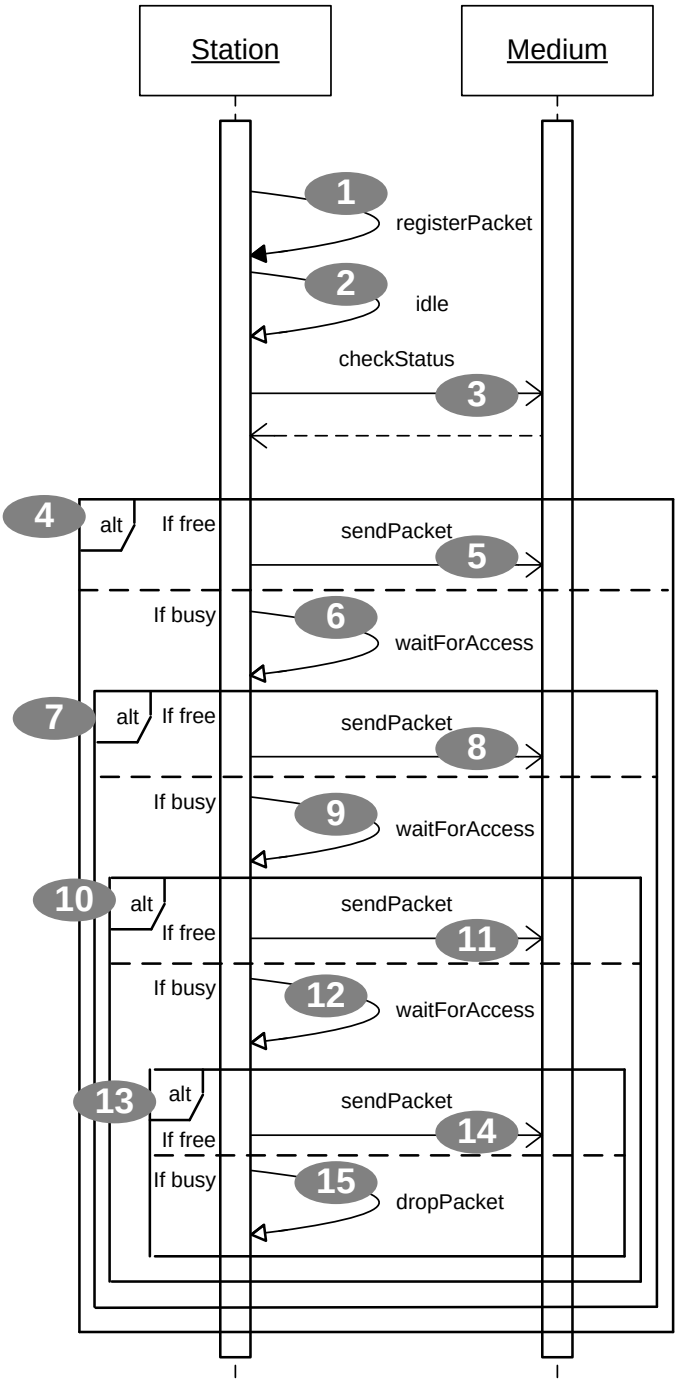

Figure 31: Sequence Diagram for a station in PAN

5.4.4 Petri Net Representation of the Scenario Generated via SD2PN

In this section, the Sequence Diagram from Figure 31 is transformed into an equivalent Petri Net using SD2PN. To begin the model transformation process, the Sequence Diagram is first

decomposed into fragmentsas defined in Section 4.1.1. The numbers depicted in Figure 31 represent each Sequence Diagram fragment. Each fragment is transformed into an equivalent Petri Net block using the model transformation rules defined in Section 4.1.2. Once every Sequence Diagram fragment has been transformed into Petri Net blocks, the process of composing the Petri net blocks starts with the mapping of the causal relationships. This mapping requires calling the *morph* function recursively for each causal relationship in the original Sequence Diagram. Once all the causal relationships are mapped, the hierarchical relationships between the Petri Net blocks are considered. The hierarchical relationships are mapped by recursively applying the *substitute* function for every *placeholder* that exists in the Petri Net blocks. This results in an integrated Petri Net as shown in Figure 32.

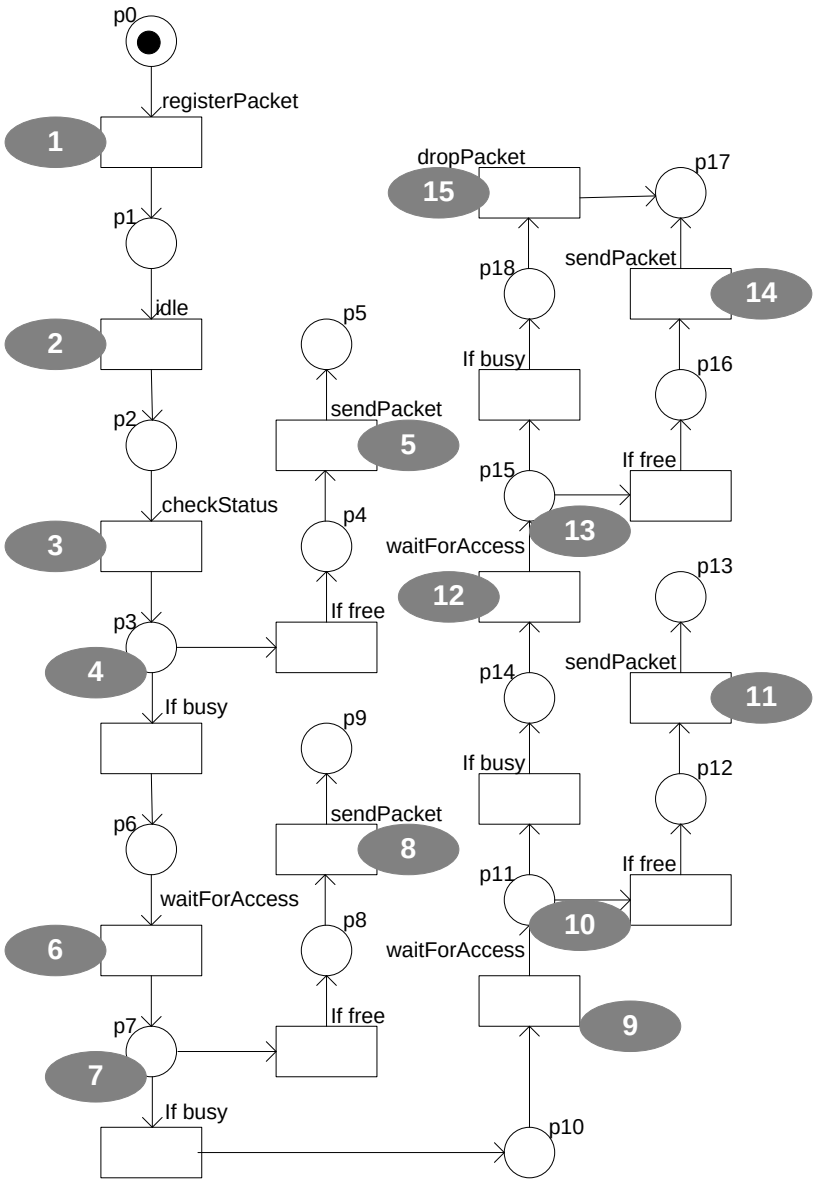

Figure 32: Petri Net for a station in PAN

5.4.5 Analysis of the Petri Net

As previously presented in Section 5.2, there are various analysis methods available in Petri Nets. In this section, the Petri Net in Figure 32 is subjected to three analysis methods; liveness, boundedness and reachability analysis using PIPE [28].

The liveness and boundedness of the Petri Net are both calculated through State Space Analysis in PIPE where the liveness is determined through the absence of deadlocks in the

Petri Net while boundedness is computed through a P-invariant calculation. The result of the analysis confirms that the Petri Net is live and bounded. Through the P-invariant calculation, it is also revealed that the Petri Net in Figure 32 is safe (bounded with the value of 1). Subsequently, a reachability analysis is performed on the Petri Net, resulting in a Reachability Graph as presented in Figure 33. As a result, the reachability analysis reveals that every state in the Petri Net is reachable through a series of event.

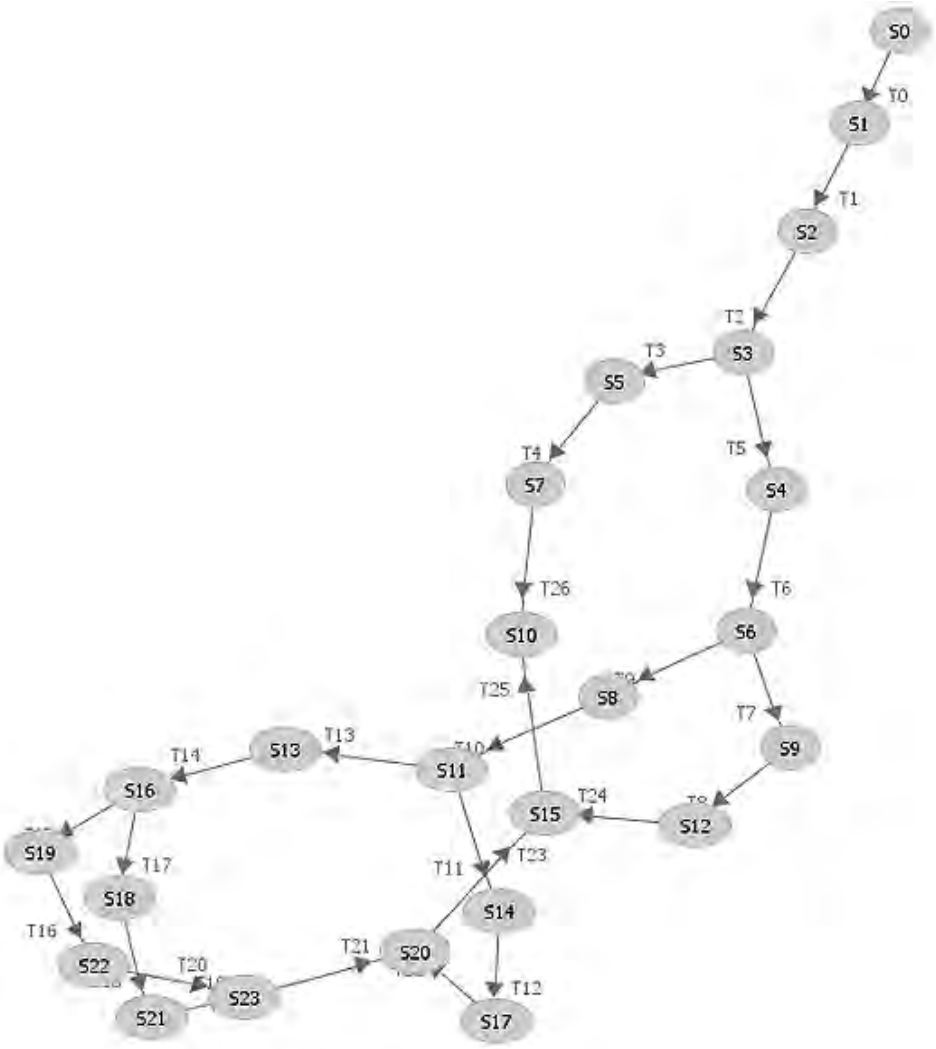

Figure 33: Reachability Graph generated using PIPE

5.4.6 Discussion

The example presented in this section highlighted the capabilities of SD2PN in transforming Sequence Diagrams into Petri Nets, allowing various structural and behavioural analyses to be performed. Although the example is fairly simple, the liveness and boundedness analysis results reveal there are no deadlocks in the design of the PAN example and that the system is safe. This is vital in ensuring that there are no unforeseen behaviours such as a deadlock causing an unexpected termination or a trap in the system causing a registry overrun. The reachability analysis also reveals that every state in the system is reachable, thus ensuring none of the stations in the PAN are left out.

However there is a limitation in the scope of analysis in general Petri Nets. There is a vital analysis component that could be applied to the example – performance analysis. In a real-time or performance oriented system, timing properties are required to perform performance or quality of service (QoS) related analysis. General Petri Nets do not have the capability to handle timing properties. However there is a class of Petri Net, Timed Petri Net (TPN) that allows timeliness properties to be integrated into the firing of transitions in the net. This paves the way for an extension of SD2PN, as presented in Chapter 7.

CHAPTER 6

SEQUENCE DIAGRAM SYNTHESIS VIA SD2PN

Complex systems are not modelled as monolithic units, but often synthesized from various smaller models. Multiple models may be used to model different scenarios, different modules or even different perspectives of a system. For example, a general model of the system may portray the core elements of a system where else different models may be used to convey the security [54] or quality-of-service (QoS) aspects of the same system. The process of integrating two or more such models of a system is referred to as synthesis. In software design, Pfleeger and Atlee [143] define synthesis as the process of building a large structure (e.g., a software design) from smaller building blocks. Agerwala and Choed-Amphai [59] define synthesis as a method to integrate complex systems by putting elements together according to a set of pre-determined rules and constraints. The benefits of bridging the gap between model design and synthesis is also described in Section 3.2.3.3.

This chapter addresses the notions of synthesis in Sequence Diagrams and Petri Nets independently and how Multi Paradigm Modelling and Petri Nets, via SD2PN inspire three

synthesis algorithms for Sequence Diagrams. This chapter also features a case study of an e-business model that is built incrementally using the aforementioned synthesis algorithms. The sections that make up this chapter are 6.1 and 6.2 addressing the notion of synthesis in Sequence Diagrams and Petri Nets respectively, while section 6.3 presents Petri Net inspired synthesis methods in Sequence Diagrams, with three algorithms for automated synthesis.

6.1 Synthesis in Sequence Diagrams

There have been a few approaches to Sequence Diagram synthesis, including Liang et al. [43], who describe a method for integration of Sequence Diagrams based on formalization of the Sequence Diagram into typed graphs. The method presented in their paper is designed for Sequence Diagram consisting of lifelines and messages. However based on [44], this synthesis algorithm does not include complex constructs such as parallelisms and conflicts, as evident in page 133 of [44]. Bowles and Bordbar [45] present a method of synthesis by mapping a design consisting of multiple views modelled by Sequence Diagrams into a unique mathematical model which is used for analysis and detecting inconsistencies. In [8], Sequence Diagrams are synthesized manually in order to create Alloy [5] models for analysis of the system. However, manual synthesis of Sequence Diagrams are error-prone, tedious and redundant. In Aspect-Oriented Modelling, the sequential composition of aspects could also be regarded as synthesis. The same also applies to the weaving of scenarios using an automated aspect weaver as described in [46]. The concept of synthesis also exists in Message Sequence Charts (MSC), a predecessor of Sequence Diagrams. Krüger's [47] approach for refinement of MSC where a notational semantics for MSC features a notion of message refinement,

where a message is syntactically replaced by a protocol for every occurrence in the MSC. However, this approach of refinement does not preserve the equivalence relations as stated in page 172 of [47].

The need for automated support for model synthesis has been outlined in various works[8, 43-47, 55-57]. This is to avoid the possibility of undesirable emergent properties from performing ad-hoc synthesis. In addition, essential properties of the individual models may not be preserved if thesynthesis is performed in an incorrect manner.

6.2 Synthesis in Petri Nets

The notion of synthesis in Petri Net is a well-studied area of research [12, 17, 20-26, 48, 58, 59, 144-146]. Some of the methods used in Petri Net synthesis include top-down [22, 23, 48], bottom-up [23, 48], hybrid [26], knitting technique [20, 21, 25, 144], reduction [145, 146] and rough set [24]. Here the two well-studied methods for Petri Net synthesis are described; top-down synthesis method and bottom-up synthesis method.

6.2.1 Top-Down Synthesis Method

A top-down synthesis method commonlybegins with an aggregate model of the system that disregards low-level details. Using the aggregate net as the basis, low-level details are gradually added into the net using a refinement process.

One method of refinement is outlined by Valette[16], which focuses on the transitions in the Petri Net. This method replaces a transition with a different, complete Petri Net. For example, assume a Petri Net *N*has a transition *t*and another Petri Net *T* that consist of low

level details for the transition *t*.Using transition refinement, the transition *t* in the net *N* can be replaced with the net *T*, where all the input arcs into *t* and the output arcs from *t* are redirected into the start and end places of *T* respectively.

Another method used in this technique is place refinement outlined by Suzuki and Murata[147]. In this method, the place that needs to be refined, p_0 is first replaced with two places p_1 and p_2 and a transition t_0. The input of p_0 is redirected as input for p_1 and the output of p_0 is redirectedas the output of p_2. The transitiont_0 must have only $p_1$as the input place and only$p_2$as the output place. Themarking of p_1 should be the same as the marking for $p_0$while the marking of p_2 must be 0. This is followed by performing a transition refinement ont_0.

The refinement technique has since been adopted by many including [22, 23, 26]. In essence, this technique can also be extended to replace a specific block of Petri Net (such as an SD2PN *message* block in Figure 34 (a), refer Section 4.1.2.1) with a different Petri Netas depicted in Figure 34 (b).

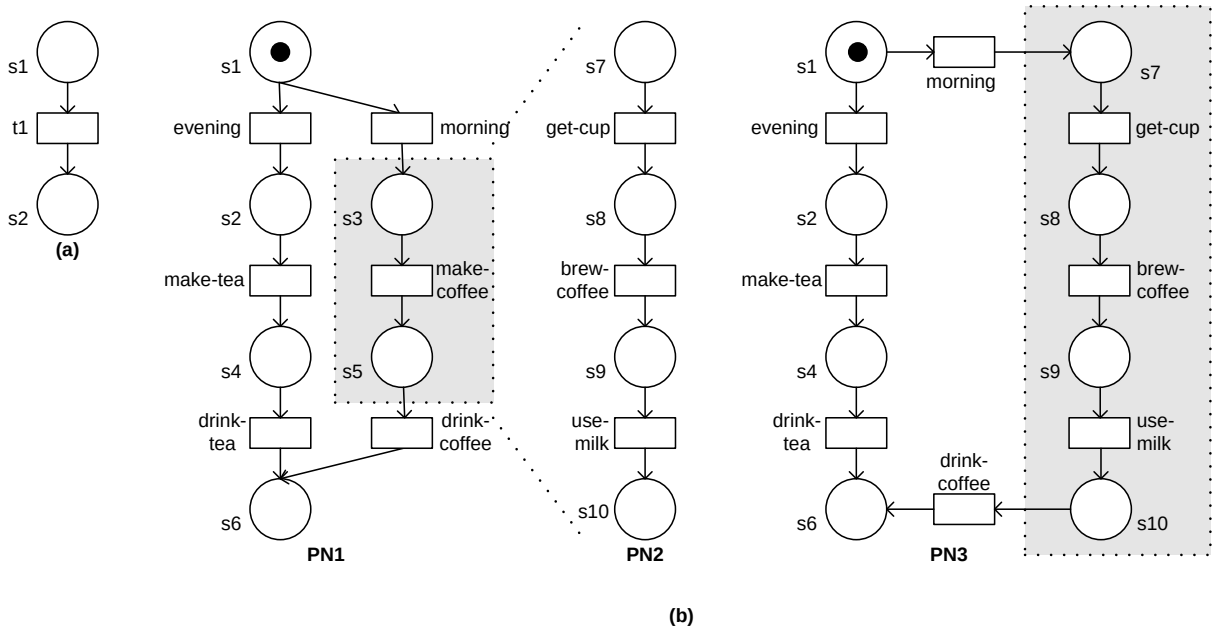

Figure 34: Example of top-down synthesis in Petri Nets

The intentionally trivial synthesis example in Figure 34 (b) presents two Petri Nets PN1 and PN2 such that PN1 describes the daily morning and evening routines of the author and PN2 that describes how to make coffee. By performing a top-down synthesis using the refinement technique, the both PN1 and PN2 could be put together; where the block of Petri Net that describes the action '*make-coffee*' is refined with the entire Petri Net *PN2* resulting in the Petri Net *PN3*. Of course it is also possible to refine the action '*make-tea'* similarly, but for the purpose of this example, it is assumed that the reader already knows how to make tea.

6.2.2 Bottom-Up Synthesis Method

The bottom-up synthesis method is commonly used to integrate two or more Petri Nets that contain common nodes. This is done by merging of places and transitions between the Petri Nets while preserving all the interactions between the nodes.

One common technique used in the merging of places between Petri Nets is called *one-way merge*[59]. Using one-way merge, a set of common places between the Petri Nets are merged to form a new place. For example, suppose a set of places to be merged, S_m are merged into a new place p. For each place such that $s \in S_m$, the input arcs and output arcs for s, ${}^{\circ}s$ and s° respectively must be represented in ${}^{\circ}p$ and p°.

As a continuation to the example presented in Figure 34 (b), the example in Figure 35 is used to present bottom-up synthesis in Petri Nets in an easy-to-understand manner.

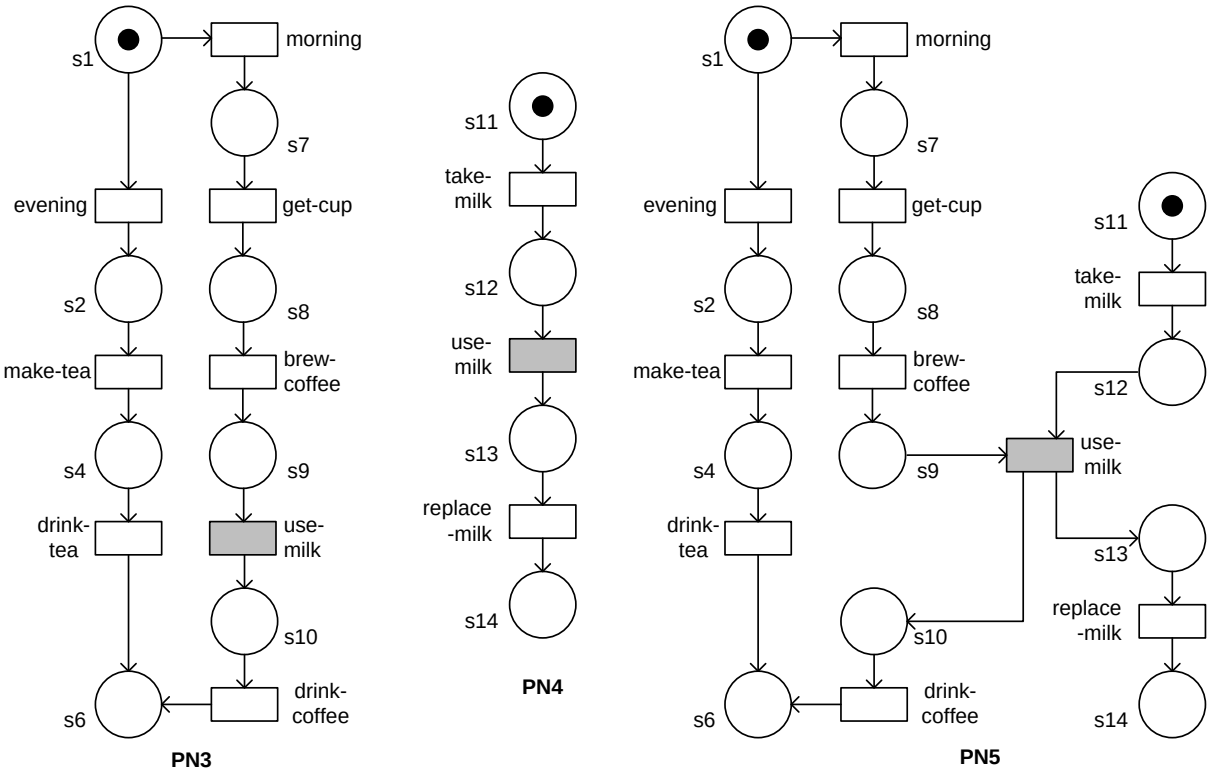

Figure 35:Example of bottom-up synthesis in Petri Nets

Suppose *PN3*[8]in Figure 35 still represents the morning and evening routines of the author while the Petri Net *PN4* describes a household policy when it comes to using milk. The two Petri Nets describe two different viewpoints and may exist independently of each other. However based on the common element between the two Petri Nets, '*use-milk*', *PN3* and *PN4* can be synthesized to create a Petri Net *PN5*.

6.3 Petri Net Inspired Synthesis of Sequence Diagrams

It is clear from sections 6.1 and 6.2 that the notion of synthesis is better established in the formal mathematical language of Petri Nets as compared to UML Sequence Diagrams.

[8] The Petri Net *PN3* in Figure 35 is the same as the Petri Net *PN3* in Figure 34(b) with only cosmetic alterations.

However, using Multi-Paradigm Modelling, it has been established in Chapter 4 that Sequence Diagrams could map into Petri Nets without losing any semantic properties via SD2PN. As such, this section presents synthesis methods for Sequence Diagrams, inspired by Petri Nets.

As means to illustrate the capabilities of the Sequence Diagram synthesis methods, as well as to provide better understanding of the algorithms, a case-study of an e-business model is used throughout this chapter – with each algorithm building up the model incrementally from the basic model in Figure 36.

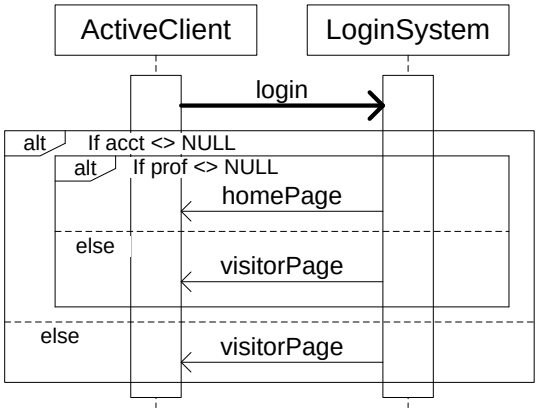

Figure 36: Basic e-Business Model

The Sequence Diagram synthesisalgorithmsintroduced in this chapter are presented in the form of pseudocodes. The following notations are used for the purpose of the pseudocodes.

Notation 1:

L: set of lifelines in the Sequence Diagram

Suppose the Sequence Diagram in Figure 36 as *SD1*, the set of lifelines in *SD1*can be written as L_{SD1} = {*ActiveClient*, *LoginSystem*}.

Notation 2:

E: set of all events in the Sequence Diagram

Suppose the Sequence Diagram in Figure 36 as *SD1*, the set of all events in *SD1* can be written as E_{SD1} which would comprise of the sending and receiving of all messages in *SD1*.

Notation 3:

M: set of messages in the Sequence Diagram

A message $m \in M$ is a tuple (*a*, *s*, *r*)indicating the operation signature *a*, the sender lifeline s, and the receiver lifeline r, such that $s, r \in L$. A message *m* also has direct associations with its sending and receiving events, denoted by *m.snd*, *m.rcv*.

Notation 4:

Scope(*m*): list of the nested combined fragments containing the message *m*, ordered from the inner to the outer fragment.

The function *Scope*(*m*) identifies the specific location of the message *m* with regards to combined fragments. Suppose the Sequence Diagram in Figure 36 as *SD1*;

$$Scope(login) = SD1$$

since the message *login* is not contained in any combined fragments. However

$$Scope(homePage) = SD1.alt(2)\#1.alt(2)\#1$$

where the message *homePage* resides in a nested *alt* fragment. *SD1* refers to the Sequence Diagram, *SD1*.*alt*(2)#1 refers to the combined fragment *alt*with two (2) fragments in *SD1*; and the fragment number one (1),and *SD1*.*alt*(2)#1.*alt*(2)#1 refers to the *alt* combined fragment with two (2) fragments that resides inside another *alt* with two (2) fragments; and that the message resides in fragment number one (1) of the nested *alt* combined fragment.

Using Notations 1 through 4 as presented above, each Sequence Diagram synthesis method introduced in this section is presented in the form of a pseudocode, and a case study example of an e-business model. Furthermore following each case study example, a complementary example of Petri Net synthesis of the same models is presented as a method for comparison between the new Sequence Diagram synthesis techniques introduced in this section and the well-studied Petri Net synthesis techniques from [16, 59].

6.3.1 Top-DownSynthesis Method in Sequence Diagrams

The top-down synthesis method for Sequence Diagrams refers to composing low-level details into an aggregate model by replacing a single message with an entire Sequence Diagram.This method allows for a Sequence Diagram that represents a top-level view of the system to be refined by multiple other low-level Sequence Diagrams in order to provide an integrated view of the system. The inspiration for this method comes from the transition refinement technique in Petri Net described in Section 6.2.1. Figure 34 also presented an example of this technique where a Petri Net block (that refers to a Sequence Diagram message as established in SD2PN Rule 1) is refined using an entirely different Petri Net.

To maintain a naming relationship between the synthesis methods of Petri Nets and Sequence Diagrams, the technique presented in this section is called *message* refinement. A

message refinement synthesis integrates two Sequence Diagrams, *SD1* and *SD2* by replacing a message *m* in *SD1* with an entire Sequence Diagram *SD2*.The algorithm for message refinement in Sequence Diagram is as follows:

INPUT: (*SD1*, *SD2* and *m)* where *SD1*and*SD2* are two Sequence Diagrams and *m* is a message in SD1 that appears only once[9]. There must also not be a copy of the message *m* in *SD2*.

OUTPUT: *SD3*, where every execution trace of *SD3* consists of an execution trace in *SD1* in which the message *m* is substituted with an execution trace of *SD2*. For every execution trace in *SD1* with exception of the message *m*, there exist an execution trace in *SD3* where the trace of *SD1* appears and for every execution trace in *SD2* there exist an execution trace in *SD3* where the trace of *SD2* appears. In addition, the message *m* does not appear in any execution traces of *SD3*.

CONDITION: Suppose *f* represents the first message in SD2 (such that there are no other event occurrences *beforef*) and *l* represents the last message in SD2 (such that there are no event occurrences *afterl*)[10]; the lifeline that sends the message *f* must be the same as the lifeline that sends the message *m* in *SD1*, and the lifeline that receives the message

[9] Should there be need to refine multiple instances of the same message, the algorithm should be run repeatedly; each time refining a single instance of the message *m*.
[10] The terms *before* and *after* here refer to the General Ordering of events in Sequence Diagrams (refer Section 2.1.1).

l must be the same as the lifeline that receives the message *m* in *SD1*.

ALGORITHM:

1. Copy *SD1* to *SD3;*
2. Identify the sender, receiver and scope of message *m* as*m.snd*, *m.rcv* and Scope(*m*) respectively;
3. If there exist a set of lifelines in *SD2* that does not exist in *SD3*, copy the aforementioned set of lifelines into *SD3* such that the set of lifelines in *SD3* consist of all the lifelines in *SD1* and *SD2*;
4. Remove the message *m* from *SD3*;
5. Add all the messages and Combined Fragments from *SD2* into *SD3* ensuring that they correspond to their respective lifelines in the previous location of*m*, such that (recalling the definition of *f* and *l* from the CONDITION section) the message *before f* must be the same as the message *before m* in *SD1* and the message *afterl* must be the same as the message *after m* in *SD1*;
5. End.

To illustrate the message refinement method with an example, a top-level Sequence Diagram is presented in Figure 37 (a) and a low-level Sequence Diagram is presented in Figure 37 (b) describing a basic e-business model and a refinement model respectively.

Using the message refinement algorithm, the message 'login' in Figure 37 (a) is refined with the Sequence Diagram in Figure 37 (b). The application of the message refinement method results in a Sequence Diagram as shown in Figure 37 (c). For illustration purposes, the messages that are involved with the refinement are drawn with lines of different thickness.

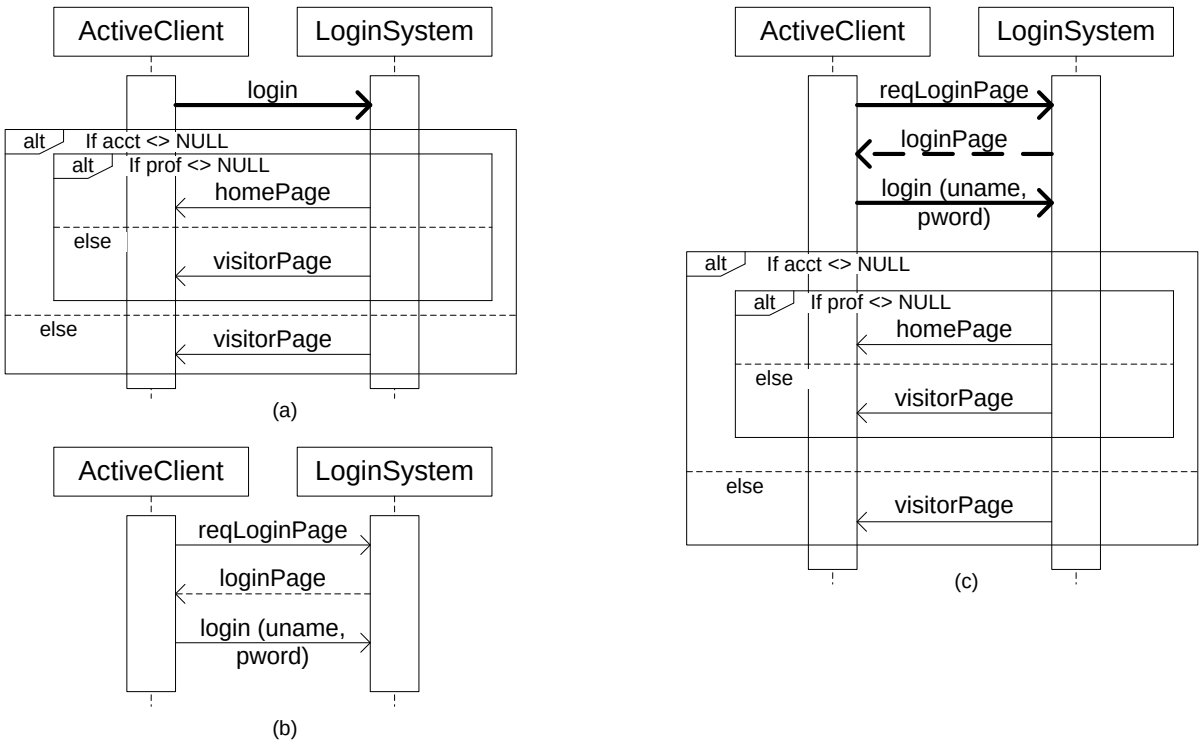

Figure 37: Example of a message refinement synthesis method featuring (a) a top-level Sequence Diagram, (b) a low-level Sequence Diagram and (c) the result of applying message refinement synthesis method to (a) and (b).

In order to compare the Sequence Diagram synthesis algorithm presented in this section with the well established top-down synthesis method in Petri Nets, the Sequence Diagrams in Figure 37are transformed by using SD2PN into the Petri Nets shown inFigure 38.

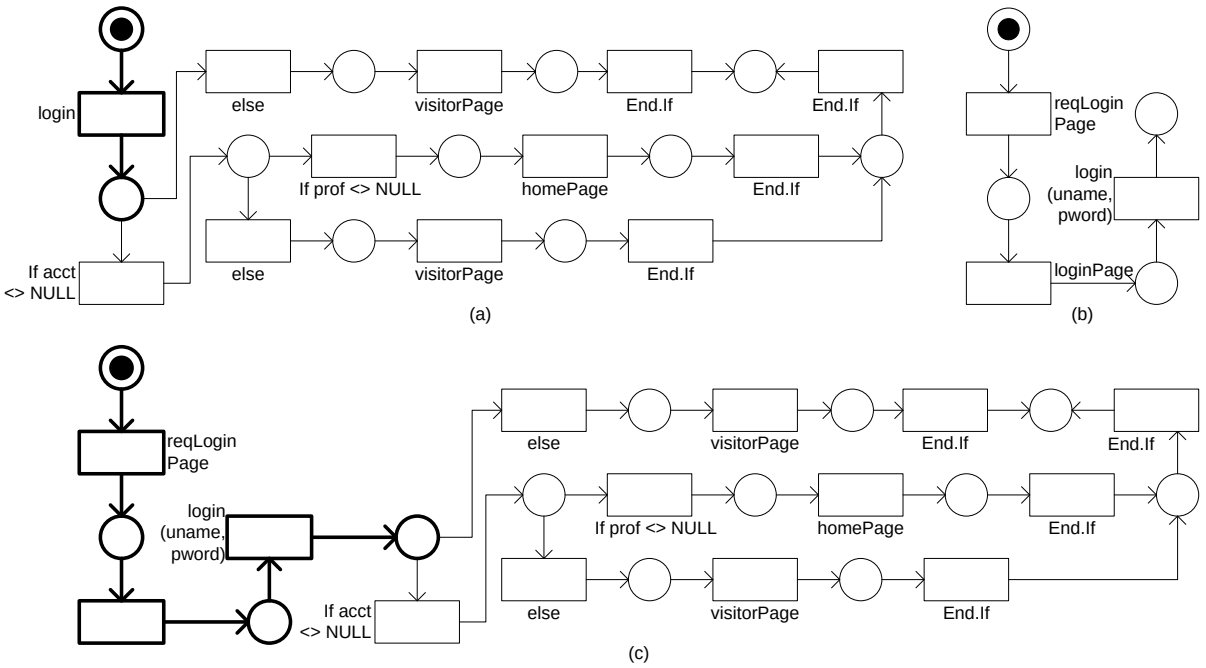

Figure 38: Petri Nets derived from Sequence Diagrams in Figure 1 (a), (b) and (c) respectively.

The Petri Nets in Figure 38 (a) and (b) are subjected to a top-down [22] synthesis method in Petri Nets where the transition 'login' in Figure 38 (a) is refined with the Petri Net in Figure 38 (b). This results in a Petri Net identical to SD2PN transformation of the Sequence Diagram from Figure 37 (c), as presented in Figure 38 (c).

6.3.2 Bottom-Up Synthesis Method in Sequence Diagrams

A bottom-up synthesis method in Sequence Diagram is ideally used to put together multiple Sequence Diagrams based on the common elements between the diagrams. As described in Petri Net synthesis method (Section 6.2.2), the bottom-up technique merges all the common elements as one element; where all the relationship are still preserved.

Figure 35 depicted two Petri Nets of different perspective, but with one common element. Performing bottom-up synthesis on the Petri Nets created a Petri Net *PN5* that

merged the common elements of *PN3* and *PN4* while still preserving all the relationship between the nodes. One observation that could be made at this point is how the actions '*drink*' and '*replace-milk*' occur non-deterministically; unrelated and independent of each other. It does not matter if the action 'drink' occurs before, after or at the same time as '*replace-milk*'. Unfortunately, it is not possible[11]to achieve this result in Sequence Diagrams. If a bottom-up synthesis method is performed via a simple merge in Sequence Diagram, there would be an emergent relationship between the two actions; i.e. if the action '*drink*' is located visually above the action '*replace-milk*', it implies a causal relationship between the two events that should not be there. The same is also true if the action '*replace-milk*' is located above the action '*drink*'. This enforces a behaviour that does not exist in the original set of execution traces, and as such does not represent a good synthesis result. One possible solution to this matter is the use of a *parallel* combined fragment. This allows both actions '*drink*' and '*replace-milk*' to occur concurrently. However, this scenario also enforces an emergent behaviour if the form of concurrency that does not exist in the original set of execution traces.

However, there are cases where the bottom-up synthesis method in Petri Nets can be translated to bottom-up synthesis in Sequence Diagrams. Two such examples of the synthesis techniques are presented in the following sections.

6.3.2.1 Part Decomposition Synthesis Method

The part decomposition synthesis method refers to replacing a lifeline in a Sequence Diagram with a complete Sequence Diagram that corresponds to it. This method allows for a lifeline,

[11] As of now, based on the author's knowledge

which represents a composite object with an internal structure, to be expanded, providing more accurate information of the entities involved in the execution of a message. The lifeline in question can be replaced with an entire Sequence Diagram, provided that all the messages into and out of the original lifeline are accounted for.

The inspiration for this method came from page 497 of [7] where UML 2.1 illustrates the existence of this capability. However, here the synthesis method is in the form of an algorithm; a Part Decomposition synthesis integrates two Sequence Diagrams *SD1* and *SD2* by replacing a lifeline *L* in *SD1* with the entire Sequence Diagram *SD2*. For the purpose of this algorithm, the notation '[]' is regarded as an array of elements. The algorithm for an automated Part Decomposition synthesis is as follows:

INPUT: (*SD1*, *SD2* and *L)*,where *SD1*and*SD2* are two Sequence Diagrams and *L* is a lifeline in SD1representing a composite object with an internal object structure and *SD2* a purpose-built Sequence Diagram that contains describes the internal object structure of the lifeline *L*.

OUTPUT: *SD3*, where the set of all lifelines in *SD3* is equal to the set of all lifelines in *SD1* and *SD2* except for *L*.

CONDITION: There should not be a duplicate of lifelines between *SD1* and *SD2*, such that $L_{SD1} \cap L_{SD2} = \emptyset$. The set of all messages in *SD2* should be a subset of the set of all messages in *SD1* such that $M_{SD2} \subset M_{SD1}$ where all the messages in and out of the lifeline *L* must be the same as the messages in and out of *SD2*.

ALGORITHM:

1. Copy *SD1* into *SD3*;
2. Identify all outgoing and incoming messages from *L* and assign them to *out* [] and *in* [] respectively;
3. Let *new_out* [] = all outgoing messages from SD2 and *new_in* [] = all incoming messages from SD2;
4. Copy the set of all lifelines from *SD2* into *SD3*;
5. For each message $m \in$ *out* []: merge m with its corresponding message $m' \in$ *new_out* [] such that the sending lifeline m'.snd and the receiving lifeline *m.rcv* is retained;
6. For each message $m \in$ *in* []: merge with its corresponding message $m' \in$ *new_in* [] such that the sending lifeline *m'.snd* and the receiving lifeline *m.rcv* is retained;
7. End.

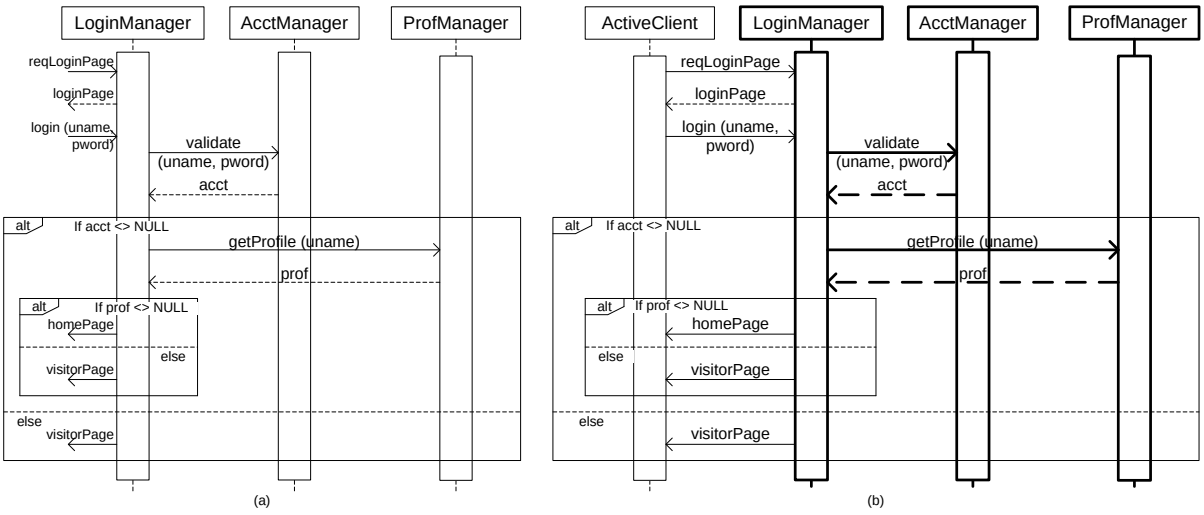

Figure 39: An example of the part decomposition synthesis method featuring (a) the internal structure of 'LoginSystem' lifeline in Figure 1 (c) and (b) the result of Part Decomposition synthesis.

To demonstrate the Part Decomposition synthesis method, a continuation of the previous example is used. The result of the previous synthesis method in Figure 37 (c) is used as the top-level Sequence Diagram for this example while the Sequence Diagram in Figure 39 (a) represents the internal structure of the lifeline 'LoginSystem' in Figure 37 (c). The application of the Part Decomposition synthesis method to the Sequence Diagrams in Figure 37 (c) and Figure 39 (a), it results in a synthesized Sequence Diagram as shown in Figure 39 (b).

To compare the Sequence Diagram synthesis algorithm presented in this section with the well established bottom-up synthesis method in Petri Nets, the Sequence Diagrams involved in the synthesis as well as the synthesis results are transformed into Petri Nets via SD2PN. In the example presented in Section 6.3.2, common messages between Figure 37 (c) and Figure 39 (a) are merged to create a new Sequence Diagram.

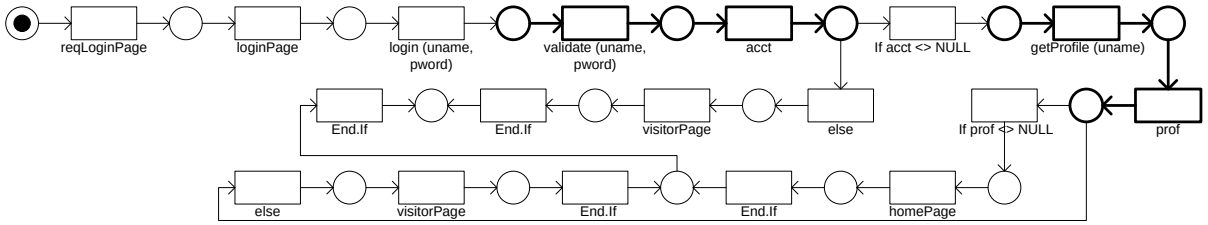

Figure 40: Petri Net derived from the Sequence Diagram in Figure 2 (b).

In this example, the application of the synthesis method generates a peculiar result. Using SD2PN, the Sequence Diagram in Figure 39 (a) can be transformed into the Petri Net in Figure 40. By applying the bottom-up synthesis method between the Petri Nets in Figure 38 (c) and Figure 40, it still results in the Petri Net in Figure 40. This is true because every node in the Petri Net in Figure 38 (c) is already present in Figure 40. As peculiar as this result is, it still matches the Sequence Diagram in Figure 39 (b).

6.3.2.2 Special Case Method: Synthesizing Attack Scenarios

In this section, a synthesis method of a different nature is introduced. This is a domain-specific method to simulate security breach scenarios where a specific attack model (i.e. man-in-the-middle attack) is composed with a given scenario. Suppose a regular scenario model in *SD1* and the behaviour of an attacker in *SD2*. The resulting Sequence Diagram will depict an act of security breach on the system and can be used to analyze the possibility of an attack as well as counter-measures. This method is motivated by the challenges faced in [54]. This synthesis enforces a special condition regarding the preservation of the sending and receiving events on certain lifelines (i.e. the parties whose communication is intercepted by the attacker). Any direct message *m* between these two parties in *SD1* is replaced by a couple of messages *(m',m")* with the same signature, flowing through the attacker in *SD2*. We shall refer to *m, m'* and *m"* as *matching* messages.

The special case synthesis method creates a Sequence Diagram *SD3* that consist of all the lifelines from *SD1* and *SD2* while replacing each message *m* between lifelines *x* and *y* in *SD1* with a couple of messages *(m',m")* from *SD2*. The algorithm for this special case bottom-up synthesis is as follows:

INPUT: (*SD1*, *SD2*, *x* and *y)*,where *SD1*is a Sequence Diagram representing a normal scenario, *SD2*is a Sequence Diagram representing the behaviour of an attacker and *x* and *y* the two communicating lifelines in *SD1* that the attacker would like to eavesdrop.

OUTPUT: *SD3*, where every execution trace of *SD2*and every execution trace of *SD1* except ((*m.snd* = *x* & *m.rcv* = *y*) or (*m.snd* = *y* & *m.rcv* = *x*)) is represented in *SD3*.

CONDITION: The number of lifelines in *SD2* must be exactly three such that L_{SD2} = {*x, y, z*} where (*x* ∈ L_{SD2} = *x* ∈ L_{SD1}), (*y* ∈ L_{SD2} = *y* ∈ L_{SD1}) and *z* representing the attacker. For every message *m* in *SD1* such that (*m.snd* = *x* & *m.rcv* = *y*) or (*m.snd* = *y* & *m.rcv* = *x*), there exist two messages *m* and *m'*in *SD2* such that ((*m.snd* = *x* & *m.rcv* = *z*) & (*m'.snd* = *z* & *m'.rcv* = *y*)) and ((*m.snd* = *y* & *m.rcv* = *z*) & (*m'.snd* = *z* & *m'.rcv* = *x*)) respectively.

ALGORITHM:

1. Create a new Sequence Diagram *SD3*;
2. Add the union of the set of lifelines in *SD1* and *SD2* to *SD3* such that $L_{SD3} = L_{SD1} \cup L_{SD2}$;
3. Identify the set *R* of direct messages between *x* and *y* in *SD1* such that*R*= {*m*| *m*=(*a,x,y*)}; each *m* being replaced by a couple of matching messages *m=(a,x,z), m' =(a,z,y)* in *SD3*;
4. For each message *m*∈ *R*, if *Scope*(m)= *Q* and *Q* ≠ *SD1*, then add *Q* to *SD3*;
5. Add all messages $m \in M_{SD2}$ into *SD3*;
6. Add all remaining combined fragments from *SD1* into *SD3*;
7. Add all messages *m* such that ($m \in M_{SD1} \& m \notin R$) into *SD3*.
8. End.

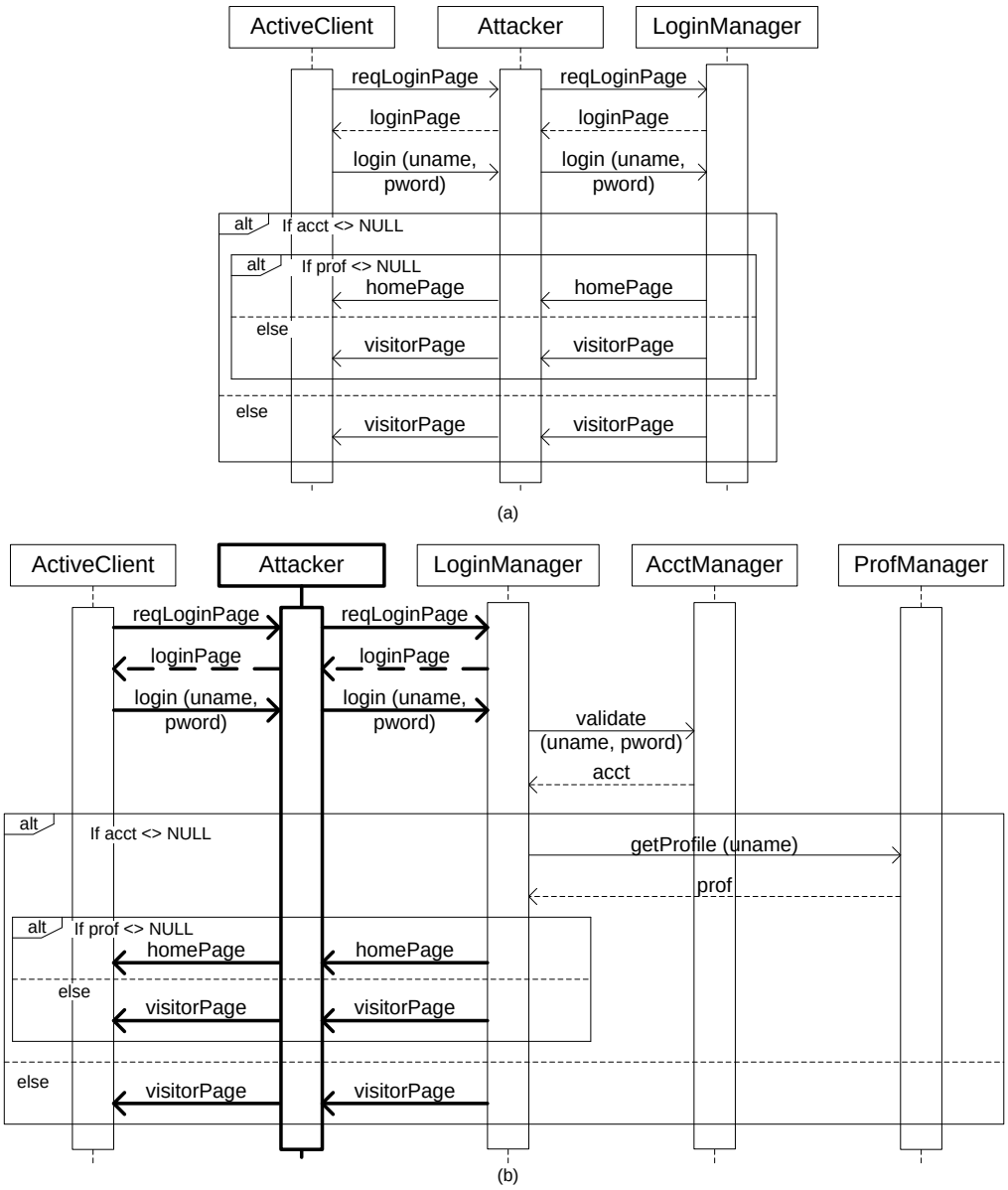

Figure 41: An example of the special case synthesis method featuring (a) the behaviour of an attacker and (b) result of the special case synthesis.

To illustrate the special case synthesis method, this example introduces an 'attacker' to the previously created e-commerce login model in Figure 39 (b). The behaviour of this attacker is depicted in Figure 41(a). The application of the special case synthesis method to the Sequence Diagrams in Figure 39 (b) and Figure 41 (a) results in the Sequence Diagram represented in Figure 41(b).

To compare this domain specific synthesis technique in Sequence Diagram with the established bottom-up method in Petri Nets, all the Sequence Diagrams involved in the synthesis are transformed into Petri Nets via SD2PN. The application of the bottom-up synthesis method to the Petri Nets in Figure 40 and Figure 42 (a) results in a Petri Net identical to Figure 42 (b); which is derived from the Sequence Diagram in Figure 41 (b).

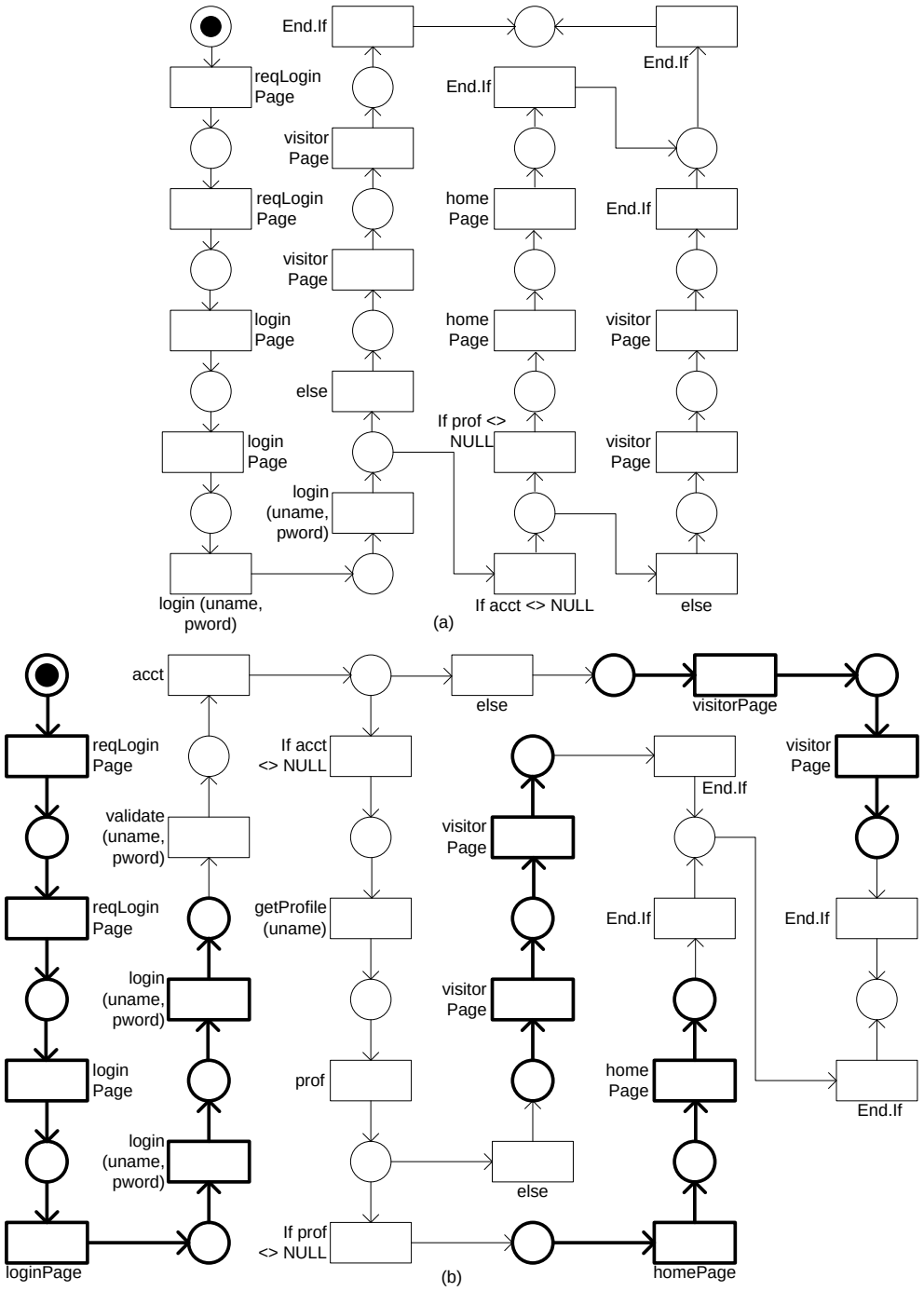

Figure 42: Petri Nets derived from Sequence Diagrams in (a) Figure 3 (a) and (b) Figure 3 (b).

CHAPTER 7

SD2PN AND TIMELINESS PROPERTIES

The notion of time in modelling and analysis is a well studied notion. This chapter presents an approach to integrate timeliness properties into SD2PN.

7.1 Significance of time in UML

It has been established that UML [7] has been offered a privileged role in the modelling community. The ability to model from an abstract view of the system, to the more functional aspects of a system such as dependencies as well as communication between objects in the system allow UML to earn a steadfast reputation as the standard in modelling. However as the number of real-time and embedded systems increase, the demand for UML to model non-functional properties, in particular timing information increases as well.

The various diagrams in UML allow structural and behavioural properties of a system to be modelled to a certain degree of accuracy. More complex properties of a system

interaction such as conflicting or parallel behaviour can also be modelled using high level additions into UML such as Interaction Operators. Restricting an interaction or behaviour in the model to reflect a specific property of the system can also be performed by integrating OCL [116] constraints into the UML model. However in this chapter, the restriction of behaviour in the UML model is narrowed down to a specific constraint; time.

Assigning timing properties to a specific event or interaction in the model could allow accurate depiction of the system in real-life where different tasks may require varying periods of time to complete. The addition of the timing constraint could change the sequence of events in the model, enforce delays between events to mimic real-life scenario as well as present emergent behaviour from the model. Analysing these properties and emergent behaviour in the modelling phase could prevent costly errors in the development phase.

In [148], the authors present a guideline on the requirements for timing constraints to be added into UML. To accurately express the time related aspects of a system, the notations used to represent the timing constraint must allow requirements such as system (or component) delays to be attached to the model. The notation must also allow external assumptions such as response time or inter occurrence times to be attached to the model. Assumptions on the underlying execution platform such as the execution times of various tasks as well as the dependencies between tasks must also be allowed as a notation on the model. Finally system behaviours that are dependent on time should be modelled on the system in the form of an independent timer or by granting access to the system clock.

In the next section, a short review of the literature is presented.

7.1.1 Review of UML extensions to include time

Although there have been various research projects down the years to integrate time into UML, the execution between the pieces of research differs vastly. The element of time in UML models is usually expressed as a time constraint. According to [149], there are three types of commonly used time constraints; hard, soft and firm. A hard time constraint demands that any event or interaction that is attached to it occurs within the constraints. Any delay in the execution is unacceptable and the system run is deemed a failure. A soft time constraint on the other hand allows violation in terms of the actual computation up to a certain degree. Finally a firm time constraint is a combination of hard and soft time constraints where the requirements are assessed as hard constraints and the computations are designated as soft constraints.

The execution of a system also relies on the arrival pattern of the messages between instances. This should also be reflected in the model. There are two major classifications in arrival patterns, *periodic* and *aperiodic*. Periodic messages have a predefined pattern. However, slight variations in the pattern are common and referred to as *jitter*. According to [150] jitter, which is a variation of *latency* or the time between two signals, transpires when there are multiple occurrence of the same signal. Aperiodic messages on the other hand do not have a pattern, and as such usually calculated using an average interval and standard deviation. Messages that are *bursty* or grouped together are also considered aperiodic messages. However bursty messages have a Poisson distribution, therefore do not have a standard deviation.

The table below presents a classification of various research projects that integrates time into UML based on several criteria.

Table 9: Review of existing pieces of research that integrates time into UML

Types of time constraints	Hard	[148], [149], [151], [152], [153], [154], [155], [49], [156], [157], [158]
	Soft	[149], [151], [156]
	Firm	[149]
Types of formalizations used	Temporal Logic	[159], [153]
	Purpose-built Formalisms	[160], [156], [157], [158]
	None	[149], [151], [49]
Attachment of time to the model	Message	[151] , [49]
	Lifeline	[156]
	Event	[148], [151], [152], [153], [155], [160], [49], [156], [157], [158]
Representation of time	Interval	[148], [151], [152], [159], [154], [160], [49], [156], [157], [158]
	Duration	[148], [151], [155], [156], [157], [158]
Classification of arrival time	Periodic	[149], [151], [153], [155], [158]
	Aperiodic	[149], [158]
Purpose of time extension in the model	Modelling real-time systems	[149], [151], [153], [161], [160], [157], [158]
	Analysis	[148], [152], [159], [154], [155], [49], [156]

All the literature presented in Table 9 uses a form of constraint to integrate timeliness properties into UML models. However, constraints are not the only way time can be represented in a model. In [161], time is presented as a scale in the model. The interval between the occurrences of events in the model is depicted by the gap between events on the

lifeline. Therefore a calculation of interval between two event occurrences could be performed by measuring the distance between the events and multiplying it by the scale. A look at page 63 of [149] assures that it is possible to allow a linear scale to be added to the UML models. However, it is an uncommon practice and to the best of my knowledge the tool support for a scaled UML model is nonexistent.

In using various forms of constraints to integrate timeliness properties into UML, most research projects choose to handle only one type of constraint; hard. This is due to the straight-forward nature of hard-constraints. In [149], it is described that most hard constraints or hard deadlines are obtained from the performance bounds of a reactive system where the system is required to react in a timely fashion to an external event. In these cases, if any of the constraints are not satisfied, the system run is deemed to be a failure. However in a real-life system, there are constraints that are relaxed to ensure less failure. Depicting the relaxed constraints (soft and firm constraints) in a model is more challenging because of the ambiguity that exists in non-absolute constraints. This is the predominant reason why a majority of research projects depict time only in the form of hard constraints.

A common approach to extending UML with timeliness properties is to formalize the model before introducing the notion of time. The classification in Table 9 reveals three different groups of formalisms; temporal logic, purpose-built formalisms or no formalism. The formalization of UML based on temporal logic allows logical statements to be inserted in the UML models as constraints; including time constraints such as deadlines, duration or interval. Besides temporal logic, other formal methods can also be used as a basis for the formalization of UML models. For example, [148, 152, 155] used automata as the basis for formalization while [154] represented constraints in the form of linear programming. However since neither automata nor linear programming contribute formalisms towards

Sequence Diagrams; they are excluded from the table. The purpose-built formalisms such as ACCORD [160], STAIRS [156, 157] and OOHARTS [158] on the other hand specify different methods of time integration into the UML models. However, research projects such as [49, 149, 151] outlines that formalizing the models is not a necessary step in integrating timeliness properties into UML models.

The attachment of time to the UML models is often presented in the form of time-stamps. These time-stamps are commonly attached to an event in the model, such as the sending or receiving of a message. This is evident in Table 9 where a vast majority of research projects attach timing constraints to event occurrences. However, time can also be represented as a duration constraint, tagged to the message itself. In these cases, the execution of the message must be of a specific duration as denoted in the tag. Although time stamps are not usually attached to a lifeline in the model, there is an exception as presented in Table 9. In [156], a variable *now* is placed at each lifeline in the model in order to synchronize between the lifelines.

Table 9 presents the classification of two methods of representing time in UML; interval and duration. Störrle [162] divides the concept of time in UML Sequence Diagrams into two types; the first of which is preserving the state of the system for a certain time interval while the second represents the duration for a single event to occur. Interval represents a time frame with a compulsory maximum and minimum value where the occurrence of the event attached to it must be within the maximum and minimum value [7, 163]. A duration on the other hand is defined as the temporal distance between two time instances [7, 163]. Duration consists of only one constraint value and the event associated with a particular duration constraint could only occur for the exact period of the specified constraint. A brief look at Table 9 points out research projects [148, 151, 156-158], where

both interval and duration constraints are supported. The remainder of the research projects supports only one or the other, with a slight bias towards interval.

The comparison of arrival patterns in Table 9 shows the two most common classification; periodic and aperiodic. Although many literatures do not specify the type of arrival pattern that are supported, the ones that do tend to choose periodic instead of aperiodic. The reason for this is similar to the choice of hard constraints instead of soft or firm constraints; aperiodic messages lead to ambiguity and non-absolute constraints, thus more difficult to represent in a model.

Finally Table 9 also shows that the purpose of extending the UML models with time constraints is evenly distributed between the need for modelling real-time systems accurately and the need for time related analysis.

7.1.2 UML 2.1 and timeliness properties

Based on the classification of research projects in Table 9, there are various methods to integrate timeliness properties into UML models. Most of the approaches taken introduce a type of formalism into Sequence Diagram where a typical Sequence Diagram metamodel is augmented with elements from temporal logic, automata or even purpose built formalism techniques. These approaches, although provide a certain degree of mathematical foundation to Sequence Diagrams – comes in the expense of the simplicity of the UML Sequence Diagrams.

To keep with the ease of use in UML, there exist a package in the UML 2.1 Superstructure document [7] named *Common Behaviours* which allows a more detailed behavioural definition in UML models. One subpackage of Common Behaviour is *Simple Time*.

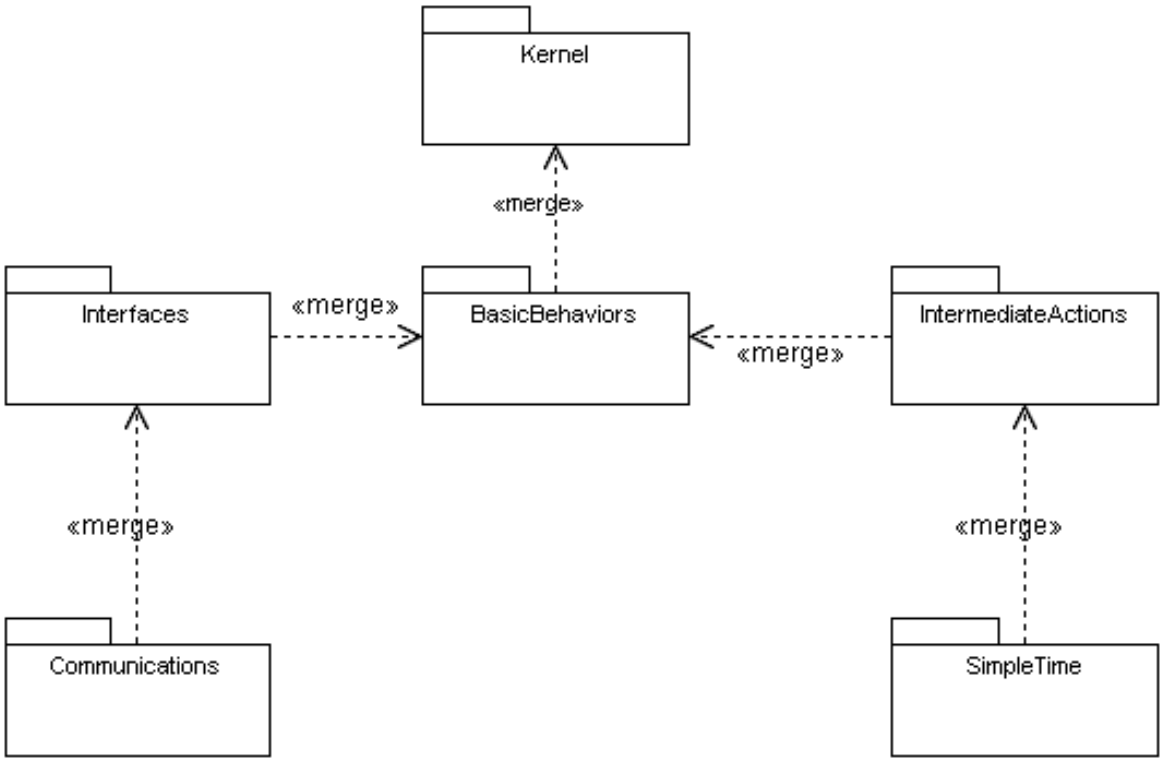

Figure 43: Dependencies between packages in Common Behaviours [7]

The Simple Time package allows time and the observation of time to be added into the UML models through the addition of relevant metaclasses. Among the metaclass additions are *Interval*, *Interval Constraint*, *Duration* and *Duration Constraint.*

Chapter 13 of [7] define Interval as the range between two values, a minimum value and a maximum value. An Interval is textually represented as the two associated values separated by "..". An example of a textual representation for Interval is:

$$< interval > ::= < min - value > '..' < max - value >$$

An Interval Constraint on the other hand is the constraint that describes the Interval it specifies. A Duration as described in page 437 of [7] specifies the temporal distance between two instances. Duration is represented as a single value that denotes the value of the duration,

commonly represented as a non-negative integer. Meanwhile the metaclass Duration Constraint defines the constraint that specifies the Duration.

There are various other metaclasses that exist in the Simple Time subpackage. However, only four relevant metaclasses are presented in this section. Although the Simple Time package allows time to be integrated into UML models, more focussed packages that deal with specific extensions of time could be added in the form of UML packages.

7.2 Extension of SD2PN to include timeliness properties

As established in the previous section, extension of timeliness properties in UML is becoming a significant factor in model design. It is also established from Table 9 that a significant number of work in this area is geared towards analysis of the time annotated UML diagrams. In this section, the SD2PN model transformation is enhanced to allow time constraints from Sequence Diagrams to be transformed into Petri Nets for the purpose of analysis.

Referring to the review of existing research in Table 9, there are various formalisms and types of constraints used in decorating the UML models with timeliness properties. In Sequence Diagrams, time constraints can be attached to an event, a message or even a lifeline. There are even different representations of time as well as the classification of message arrival. Each research presents a different approach, with different sets of requirements. Although any of the aforementioned pieces of research could be adopted as the enhancement to the Sequence Diagram metamodel in SD2PN, the enhancement presented in this section features the timeliness properties from the UML 2.1 metamodel.

In the next section, the Sequence Diagram metamodel used in the SD2PN model transformation will be enhanced to include timeliness properties from UML standard. Subsequently, the Petri Net metamodel and the SD2PN model transformation rules will also be enhanced with timeliness properties to allow the model transformation from time annotated Sequence Diagrams into Timed Petri Nets.

7.2.1 Sequence Diagram metamodel enhancement

To allow time constraints to be present in Sequence Diagrams, the Sequence Diagram metamodel in as Figure 3 is enhanced with time constraints. Figure 44 presents an enhanced metamodel for Sequence Diagrams where the shaded elements in the metamodel represent the extensions that signifies the addition of time properties into Sequence Diagrams. The shaded elements are adapted from "Common Behaviors", chapter 13 of the UML 2.1 Superstructure [7] as described in the previous section.

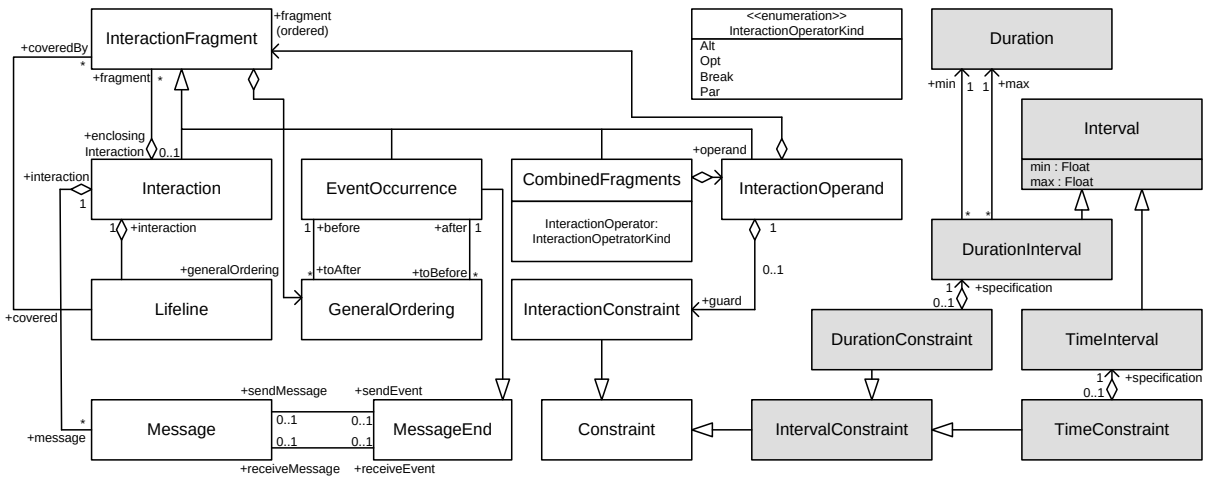

Figure 44: Sequence Diagram Metamodel augmented with Timeliness Properties

This extension of the UML Sequence Diagram metamodel used in the model transformation allows *Interval* and *Duration* constraints to be added into Sequence Diagram metamodels. This extension, or enhancement of the metamodel takes into account the various researches in Table 9 where time constraints are represented as either Intervals of Durations. Table 9 also denotes that the attachment of time to Sequence Diagram elements occurs on events, messages and lifelines – all three types of attachments are catered for in this enhancement. Finally and most importantly, this metamodel enhancement does not deviate from the simplicity, a core concept in UML. Furthermore, the metamodel itself is a subset of the UML 2.1 metamodel with no foreign model elements introduced. As such, any Sequence Diagram that conforms to the metamodel used in this research will conform to the UML 2.1 metamodel.

Both Interval and Duration are syntactically represented textually inside curly brackets as specified in [7, 163] and each value is expressed as *float* instead of *Value Specification* in order to manage the constraints more accurately and to keep the metamodel to a minimum.

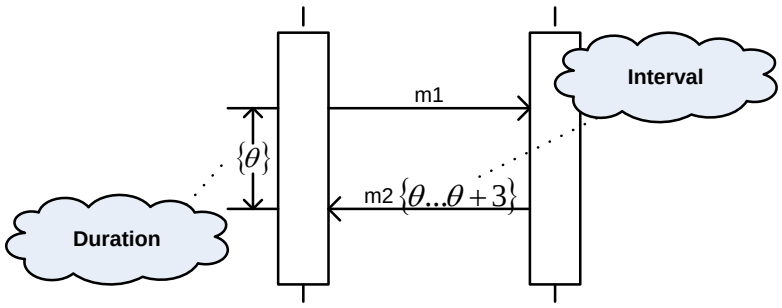

Figure 45: Example of a Sequence Diagram with time constraints

Figure 45 shows an example of a Sequence Diagram that features both types of time constraint, Interval and Duration. The Interval between the sending and receiving events of *m2* indicates that the completion (sending and receiving) of *m2* takes between θ and $\theta + 3$ to

occur, where θ is a constant. The Duration between *m1* and *m2* on the other hand indicates that after *m1* is completed, the state is preserved for the duration of θ before *m2* could be sent.

The presence of Interval and Duration in the Sequence Diagram could present a unique case that is not represented in the previously defined fragments. The example in Figure 45 shows the presence of a Duration that is not attached to a *message*. This warrants the inclusion of an additional fragment type and an additional transformation rule that will be addressed later in Section 7.2.3.

7.2.2 Petri Net metamodel enhancement

The enhancement of the Sequence Diagram metamodel with time constraints introduces an inconsistency between the source and the destination metamodels of the SD2PN model transformation. To allow the Sequence Diagrams to be accurately mapped into Petri Nets, the Petri Net metamodel has to be enhanced with time constraints as well.

The addition of constraints to an ordinary Petri Net results in a type of Petri Net called Timed Petri Net [50]. Figure 46 represents the metamodel of Timed Petri Net where the shaded elements refer to the extension of the metamodel in Figure 7 with time properties. This metamodel is compiled from the theories of time in Petri Nets from [50] as well as from a direct correlation with the Sequence Diagram metamodel. The compatibility of the new Timed Petri Net metamodel with common Time Petri Net tools has also been taken into consideration, namely PIPE and CPNTools.

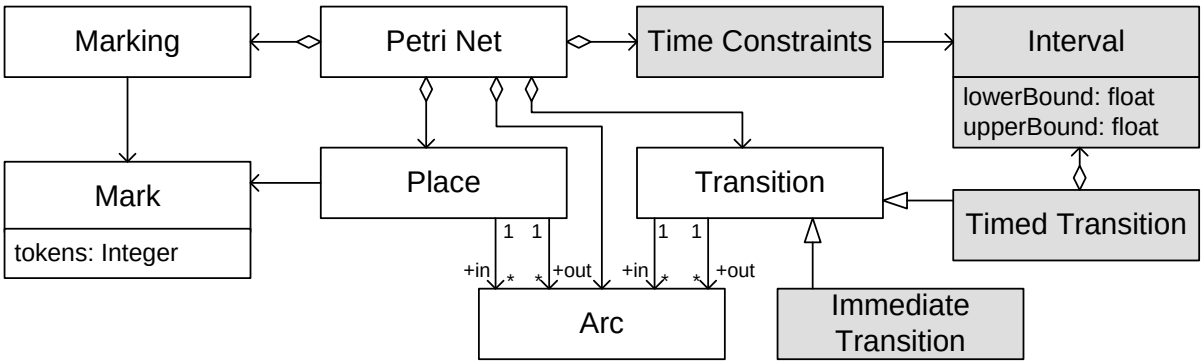

Figure 46: Petri Net Metamodel with Timeliness Properties

The shaded elements in the metamodel in Figure 46 include Interval and two specializations of *transition*; *immediate transition* and *timed transition*. The Intervals are expressed as closed intervals [50] and consists of an upper and lower bound of type *float*, to be consistent with Sequence Diagrams. Intervals are connected to *transitions*. For a *transition* to *fire*, it must be *enabled* and once *enabled*, a clock starts; the *transition* can *fire* when the value of the clock is within the interval. An example of a timed transition is shown in Figure 15 where the transition *t2* has a time constraint with the closed interval $[\theta, \theta+3]$. The transition *t2* can only fire under two conditions: it must be enabled and the clock must be between θ and $\theta+3$.

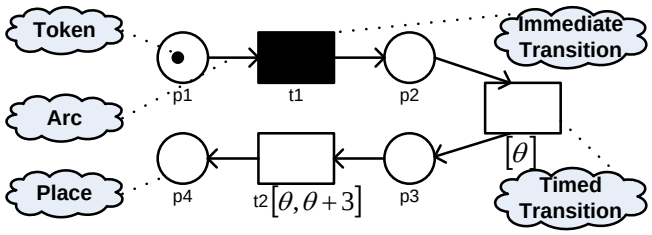

Figure 47: Example of a Timed Petri Net

Two types of *transition* are identified in Figure 47; *immediate transitions and timed transitions*. *Immediate transitions*, which are*transitions* without time constraints, are depicted

as black rectangles while the *timed transitions* are depicted as white rectangles. An *immediate transition* may be considered as equivalent to a *timed transition* with an interval of [0, 0]. For *timed transitions*, the *interval* is shown in a bracket by the label of the *transitions*, with a comma separating the upper and lower bound. If the upper and lower bound of the interval is the same, such as [50, 50], it is abbreviated as [50].

7.2.3 SD2PN Transformation Rules enhancement

In this section, the set of SD2PN model transformation rules are enhanced to include timeliness properties. Rule 1 of SD2PN is modified to accommodate the existence of the two types of *transition* while Rules 2 through 5 remains unchanged since there are no intervals or durations that are attached to CombinedFragments. Every transition in Rules 2 through 5 is therefore designated as *immediate transitions*.

Rule 1 from Section 3.2 is used to transform every message in a Sequence Diagram into a Petri Net block consisting of two *places*, *s1* and *s2*, and a *transition*, *t*. By adding a time constraint to this rule, the *transition t* is given an Interval constraint with a maximum and minimum value acting as its upper and lower bound. There are three possible cases for the execution of this rule:

Case 1: If a message has an interval associated with it e.g. {10...30}, the *transition t* in the resulting Petri Net block is designated as a *Timed Transition* with a closed interval [10, 30].

Case 2: If a message has a duration associated to it e.g. {20}, the *transition t* in the resulting Petri Net block is designated as a *Timed Transition* with a closed interval [20, 20] or abbreviated as [20].

Case 3: If a message does not have any time properties attached to it, the *transition t* in the resulting Petri Net block is designated as a *transition* with a closed interval [0, 0] or an *Immediate Transition*.

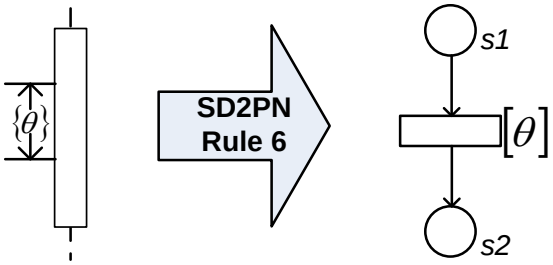

Figure 48: Rule 6 of SD2PN

Rule 6 – Duration: Recalling the new type of fragment defined in Section 7.2.1, an additional Rule is introduced to SD2PN. Rule 6, as illustrated in Figure 48 maps time properties that are not attached to any particular message into a Petri Net block. This results in a Petri Net similar to Rule 1. However, there are only two possible execution cases for Rule 6:

Case 1: If a time constraint has an interval associated to it e.g. {10...30}, the *transition t* in the resulting Petri Net block is designated as a *Timed Transition* with a closed interval [10, 30].

Case 2: If a time constraint has a duration associated to it i.e. {20}, the *transition t* in the resulting Petri Net block is designated as a *Timed Transition* with a closed interval [20, 20] or abbreviated as [20].

Despite the metamodel enhancements, transformation rules enhancements and the addition of a new transformation rule, the fundamentals of the model transformation process described in Chapter 4 remains unchanged. The three stages of SD2PN are still valid:

Stage 1: Decomposition of Sequence Diagrams into fragments.

Stage 2: Transformation of each fragment into a Petri Net block.

Stage 3: Composition of the Petri Net blocks using *morph* and *substitute*.

The process of Sequence Diagram decomposition in Stage 1 is enhanced through the introduction of an additional fragment type. In Chapter 4, five fragment types were introduced; *message* and *CombinedFragments* of type *alternative*, *option*, *break* and *parallel*. However, for the purpose of the time enhanced SD2PN model transformation, an additional fragment type is introduced, as described in Section 7.2.1.

Stage 2 of the model transformation makes use of the set of six transformation rules to transform the time augmented Sequence Diagrams into Timed Petri Nets. In Stage 3 of the model transformation, the Petri Net blocks are put together using *morph* and *substitute* to create a larger, more integrated Petri Net. The result presented in Chapter 4 where the correctness of the transformation was proved still applies to the enhanced model transformation since the enhancements made do not affect the structural consistency of the Petri Net blocks i.e. all Petri Net blocks begins and ends with a *place*. This ensures that the enhanced model transformation accurately transform Sequence Diagrams enhanced with time constraints into semantically equivalent Timed Petri Nets.

7.2.4 Discussion

In the previous section, an extension to the SD2PN model transformation to include timeliness properties is presented. The extension is based on the Simple Time subpackage from Common Behaviours, chapter 13 of the UML 2.1 Superstructure document [7]. In order to perform the extension, the Sequence Diagram metamodel is enhanced to include Simple Time metaclasses from UML 2.1. This is followed by an enhancement to the Petri Net metamodel, followed by the transformation rules enhancement.

With reference to Table 9 from Section 7.1.1, this extension technique could also be applied to other classifications of time augmented Sequence Diagrams. The Sequence Diagram metamodel could be enhanced to include various metaclasses that describes the different classifications presented in Table 9. Different aspects of the metamodel could be used to describe the different elements in Table 9 such as the types of constraints, types of formalisms or even the classification of arrival time.

To illustrate a possible extension of SD2PN, a theoretical example is presented based on Table 9 where two research projects [153, 159] are formalized using temporal logic. The logical statements that augment the Sequence Diagrams could be represented as OCL constraints [116]. As such, the Sequence Diagram metamodel only needs to be enhanced to allow OCL constraints as a metaclass. However, since Timed Petri Nets do not support logical constraints, a different subset of Petri Net must be chosen; Coloured Petri Nets [74]. Coloured Petri Nets are a subclass of Petri Nets that allow logical constraints to act as guards to the firing of transitions in the net. Finally, the transformation rules needs to be enhanced to allow the OCL constraints in the Sequence Diagrams to be transformed into guards in the resulting Petri Nets. Although the extension described in this paragraph is theoretical, it is also regarded as an area for future research and will be described in more detail in Chapter 8.

Among the classifications in Table 9, there is also a dichotomy between the reasons for augmenting Sequence Diagrams with timeliness properties. Various research projects [49, 148, 152, 154-156, 159] cited analysis as the reason for the enhancement. A common type of analysis performed in time enhanced UML models is performance analysis [49]. The next section discusses performance analysis using the SD2PN model transformation.

7.3 Using SD2PN for Performance Analysis

The extension of SD2PN with timeliness properties creates a platform for various time sensitive analysis such as performance analysis, schedulability and Quality of Service (QoS) analysis to be performed on Sequence Diagrams based on Petri Nets. In this section, an instance of the time sensitive analysis is explored; performance analysis.

The next section presents the significance of performance analysis in Sequence Diagrams, followed by the capabilities of Petri Nets in performing such analysis. Subsequently, an example of how performance analysis can be performed on Sequence Diagrams using SD2PN is presented in comparison to the structural and behavioural analysis presented in Chapter 4.

7.3.1 Significance of Performance Analysis in Sequence Diagrams

The concept of performance analysis in design models is a well studied area of research [49, 164-169]. For example, in [49], a technique for performance analysis in system design is presented based on UML Profile for Schedulability, Performance and Time Specification[170]. The UML Profile is used to tag values such as performance requirements

and resources onto UML behavioural models such as Sequence Diagrams. Although the profile is easily understandable, it lacks the level of formalism needed to perform analysis. Thus, a Layered Queuing Network (LQN) is used as the formalism for the analysis in [49]. The same sentiment is echoed in [169] where the UML Profile is acknowledged, but the analysis is performed by transforming the Sequence Diagram into a Communication Dependency Graph (CDG).

In [168], a survey on model based performance analysis in software engineering is presented, marking the transition from the traditional system development method purely concerned on the structural correctness of the system, to a more performance oriented system design. The survey outlined five major integrated methods used in model based performance analysis; Queuing Network based methodologies, process algebra based approaches, Petri Net based approaches, Simulation based methods and methodologies based on stochastic processes. Despite the five different types of approaches used, a major concern is the complexity of the performance models. Translating the design models into formal performance models requires a strong semantic mapping between the two formalisms. Even so, the complexity of the generated performance model may still prevent efficient performance analysis. Another concern highlighted in the survey is the lack of automation in the translation from the design model to the performance model, which could allow a system designer to perform the analysis and receive feedback with minimal knowledge of the formalism used in the analysis.

7.3.2 Petri Nets and Performance Analysis

While the analysis capabilities of general Petri Nets focus on the structural and behavioural properties of a system, the addition of time properties to the Petri Nets allows for performance

analysis as well. A Cycle-time analysis could be used to determine the duration for a complete sequence of action in the system while a tool such as CPNTools [27] can be used for computing the amount of time that separates two events, i.e. time between requesting access to a resource and getting the resource. Various Petri Net tools also provide a platform for other performance analysis such as average time, standard deviations, confidence intervals and throughput analysis as described in [28, 74].

7.3.3 Using SD2PN to allow Performance Analysis in Sequence Diagram

The extension of SD2PN with timeliness properties as presented in Section 7.2 allows Sequence Diagrams annotated with time constraints to be transformed into Petri Nets. The inclusion of timeliness properties in SD2PN creates a platform for time sensitive analysis such as performance analysis to be performed on Sequence Diagrams. As established in the previous sections, performance analysis in Sequence Diagrams is an active area of research. In this section, the time enhanced SD2PN model transformation is used as a platform for performance analysis in Sequence Diagrams, taking advantage of the well-established analysis methods in Petri Nets.

In Chapter 5, where the functionality of SD2PN was illustrated for the purpose of analysis, an example of the transformation process was provided. The Sequence Diagram in Figure 31, a representation of a Personal Area Network, was transformed via SD2PN into the Petri Net in Figure 32. To illustrate the introduction of time as an element in the model transformation, the Sequence Diagram in Figure 31is augmented with time constraints, resulting in the Sequence Diagram in Figure 49 (a). Using the enhanced SD2PN model

transformation, this Sequence Diagram is transformed into the Petri Net depicted in Figure 49 (b).

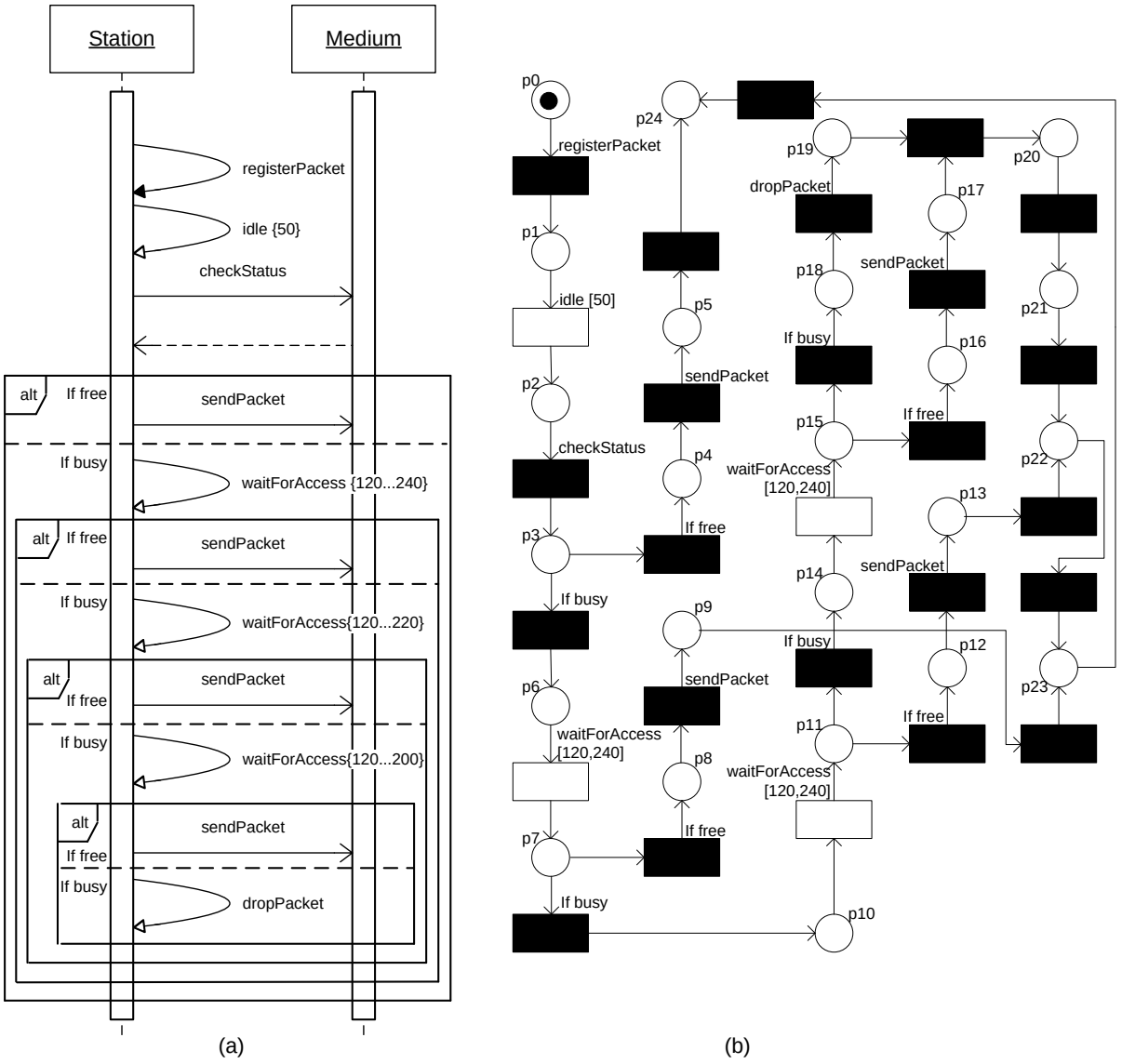

Figure 49: (a) Sequence Diagram for a station in PAN and (b) its equivalent Timed Petri Net

The Petri Net generated via the enhanced SD2PN in Figure 49 (b) is structurally equivalent to the Petri Net in Figure 32; thus indicating the consistency of the model transformation. However, the introduction of timeliness properties into SD2PN allows

performance analysis to be performed in addition to the existing structural and behavioural analysis; time-sensitive analysis such as a cycle-time, average time, standard deviations, confidence intervals and throughput analysis can be performed, as described in references [28, 74]. The focus of the performance analysis in this case is throughput analysis; this will be used to analyse the maximum delay for a station in the Personal Area Network.

The maximum delay is calculated based on the time it takes for a station to gain access to the medium *(sendPacket).* The factor that contributes to the increase in waiting time is the number of stations. A higher number of stations will increase contention between the stations. This inevitably leads to a longer maximum waiting period. For the case of a single station in the PAN, the Petri Net would be the same as the Petri Net in Figure 49 (b). However, for cases where there is more than one station, the Petri Net in Figure 49 (b) would be replicated for each station. The throughput analysis will compute the maximum waiting time based on the last station to gain access to the medium via the *message* 'sendPacket'. For example, in a case where there are two stations trying to gain access to the medium, after registering the packet (firing of *registerPacket* transition), in Figure 49 (b), both stations will face a mandatory idle time of 50μs (firing of *idle* transition) before checking the status of the medium. Following that, only one station will be able to gain access to the medium while the other will have to wait between 120μs and 240μs (firing of *waitForAccess* transition), thus a maximum waiting time of 290μs (= 240μs + 50μs).

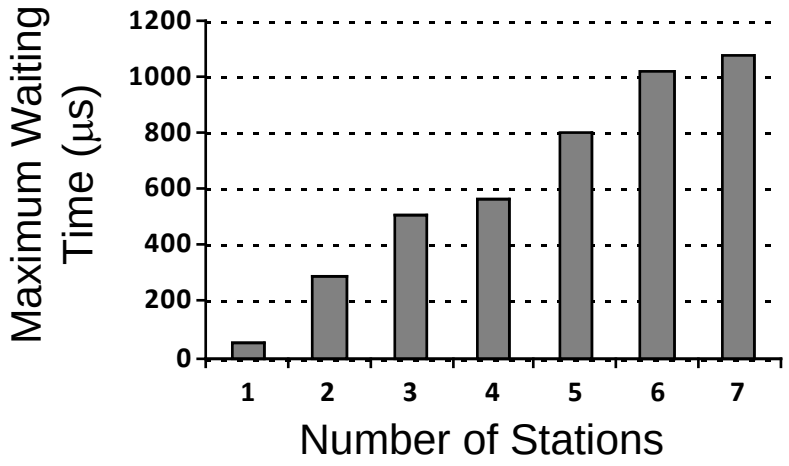

Figure 50: Maximum Waiting Time analysis result

The graph in Figure 50 indicates the maximum delay that a station may face before gaining access to the medium to send a packet based on the throughput analysis. The number of stations is limited to 7 to ensure there are no collisions; this is based on the previous assumption that the *contention window* (CW) does not increase.

In the example of the Petri Net in Figure 49 (b), the analysis performed could provide a basis to optimise related protocols to ensure a better performance. This provides a domain of interoperability from Sequence Diagrams to Petri Net allowing not only structural and behavioural analysis, but also performance analysis. The performance analysis is not limited only to throughput analysis. Various other performance analyses such as cycle-time analysis, average time, standard deviations, and confidence intervals analysis can also be performed.

CHAPTER 8

DISCUSSION AND CONCLUSION

This chapter concludes the work presented in this thesis. In Section 8.1, a summary of the contributions made in this thesis is presented. Section 8.2 presents a discussion on future work that could be done to expand and improve this research in the future while finally Section 8.3 presents the implications of this research.

8.1 Summary of Contributions

The major contribution of this thesis is presenting the application of Multi Paradigm Modelling via Model Driven Development (MDD) model transformation from Sequence Diagrams to Petri Nets. The model transformation, SD2PN is used as a vehicle to perform analysis and synthesis in Sequence Diagrams using the well-defined and well-established analysis and synthesis methods in Petri Nets.

In Chapter 2 of the thesis, an introduction to UML is presented, in particular Sequence Diagram. This is followed by an introduction of Petri Nets, including the various flavours of Petri Nets and a section on the well-studied subclass of Petri Nets; Free Choice Petri Nets. A review of existing Petri Net tools is also presented. This is followed by an introduction of Labelled Event Structures (LES) as well as the methods to translate both Sequence Diagrams and Petri Nets into LES. Finally a preliminary on MDD with a focus on a particular MDD model transformation framework, SiTra is presented.

The concept of Multi Paradigm Modelling is introduced in Chapter 3 where the role of modelling in system development is discussed together with the concept of model design, model analysis and model synthesis and the disparity between them. Besides providing an overview of how MDD model transformation, with regards to Multi Paradigm Modelling can be used to perform model analysis and model synthesis on a design model, this chapter also presents a method to overcome a weakness of Multi Paradigm Modelling which is proving semantic preservation between paradigms using a common semantics domain.

Chapter 4 presents an application of Multi Paradigm Modelling by defining a MDD model transformation from UML Sequence Diagrams to Petri Nets. The model transformation, named SD2PN, is presented in three stages; Decomposition, Transformation and Composition. In the Decomposition stage, a Sequence Diagram is split into multiple Sequence Diagram fragments based on the metamodel presented in Figure 3. In the Transformation stage, each Sequence Diagram fragment is transformed into an equivalent Petri Net block based on a set of model transformation rules which are also defined in Chapter 4. Subsequently, in the Composition stage, the Petri Net blocks are put together using two local functions *morph* and *substitute* to create a Petri Net that is semantically equivalent to the original Sequence Diagram. Chapter 4 also presents a mathematical proof that each and every

Petri Net generated via SD2PN belongs to a well studied subclass of Petri Net known as Free Choice Petri Net. Finally, to prove that the Petri Nets generated by SD2PN via the aforementioned three stages are semantically equivalent to the original Sequence Diagrams, a proof using a common semantic domain in the form of Labelled Event Structures (LES) is presented followed by another proof using mathematical notations in the form of Incidence Matrices.

Chapter 5 utilizes the model transformation, SD2PN presented in Chapter 4, coupled with the methodology presented in Chapter 3 to analyze Sequence Diagram models using well established analysis methods in Petri Nets. This chapter starts with the description of various analysis methods available in Petri Nets followed by the advantages of using Free Choice Petri Nets in performing the analysis. Subsequently, a tool for automated analysis of Sequence Diagrams using Petri Nets is presented called SD2PN Transformer. SD2PN Transformer is a Java based tool that receives Sequence Diagrams in the form of XMI and parses the XMI data to create Java objects. The Java objects are transformed into Petri Net Java objects using Java code representing the SD2PN transformation algorithm and subsequently produced as an XML file that could be read by well-known Petri Net tools. Finally to illustrate the capabilities of SD2PN, an example of a Personal Area Network (PAN) is presented, starting from the description of the protocol and the Sequence Diagram representation of the protocol up to the analysis of the protocol using Petri Net analysis methods.

In Chapter 6, well established Petri Net synthesis methods are discussed followed by the adaptation of those synthesis methods into Sequence Diagrams. Three Sequence Diagram synthesis methods are introduced in this chapter; a message refinement method, a part decomposition method and a special case method to introduce a man-in-the-middle (MiM)

type of attack into a Sequence Diagram. Each synthesis method is presented together with an algorithm, an example of an e-commerce login system and a proof of semantic preservation.

The enhancement of SD2PN with timeliness properties is presented in Chapter 7. In this chapter, the significance of time in UML is discussed followed by a review of methods to integrate time into UML. Subsequently, an extension of SD2PN to support timeliness properties is presented including the metamodel enhancements for both Sequence Diagrams and Petri Nets as well as enhancements for the transformation rules.

As an overall reflection, this thesis contributes SD2PN – an MDD model transformation that serves as a basis for Multi Paradigm Modelling between the easy-to-use, widely accepted modelling language, UML Sequence Diagrams and a formal, mathematical language, Petri Nets. This model transformation then contributes towards two directions; analysis and synthesis of Sequence Diagrams. Finally the model transformation itself is enhanced to include timeliness properties allowing for a plethora of time-sensitive analysis to be performed. The entire research is described using various examples; ranging from the intentionally trivial examples to the slightly more complicated examples that could still be followed. However this does not mean that SD2PN can only withstand case studies with limited amount of elements. Based on the synthesis methods outlined in this thesis, analysis of large-scale Sequence Diagrams can already be performed by separately analysing aggregate models before synthesizing them together – achieving scalability. However, scalability in terms of larger Sequence Diagrams is also being studied not only with the object and message parameters of Sequence Diagrams, but also the number of conflicting and concurrent behaviours. Case studies involving real-life Sequence Diagrams are being conducted and evaluated and will be a core area of study for the future research together with other areas such as described in the following section.

8.2 Future Work

This section presents the plans of the author to enhance the research further to make SD2PN a completely automated model interoperability framework that caters for analysis and synthesis of a larger range of Sequence Diagrams, including Sequence Diagrams with OCL[116] constraints attached.

The Sequence Diagram metamodel used in this research, as presented in Figure 3 is a subset of the UML metamodel derived from [7]. However, there are some elements that exist in the UML metamodel for Sequence Diagrams that has not been included in the metamodel used in this research; such as the *Combined Fragment* of type *loop* and *negative*. The *loop* operator specifies that all the messages that are a part of its operand are recurrent (looped) for a specified number of times based on the constraint(s) attached to it; while still preserving the order of causality between the messages. A *loopCombined Fragment* consists of only one operand and may contain other *Combined Fragments.*This operand may be transformed into Petri Nets by using a series of *alternativeCombined Fragments* to keep repeating a specific Petri Net block based on the condition specified in the original *loop* operand – however research is still in progress to analyze if the Petri Net block produced still preserves the semantics of the Sequence Diagram fragment; and if the resulting Petri Net could still be classified as a Free Choice Petri Net.

The *Interaction Operatornegative* in Sequence Diagrams (or abbreviated as *neg*)defines a *Combined Fragment* with only one operand where each sets of traces in the operand may not occur. Translating this operand into Petri Nets would require using a high-level addition to Petri Nets called *inhibitor arcs*. Inhibitor arcs refer to arcs such as in the Petri Net metamodel in Figure 7; however, this arc will not fire when there are tokens in the places

that are its set of inputs. At present, research is still being conducted to study the effects of inhibitor arcs to the complexity of analysis and if correctness of the transformation could still be preserved.

Plans are also being drawn-up to enhance the SD2PN Transformer (refer section 5.4). At present, the tool is a Java based tool that requires input in the form of XMI (parsed through SDMetrics into Java objects). The tool then implements the SD2PN transformation rules and generates Petri Nets in the form of Java objects which are later written into XML that could be read by Petri Net tools. In the future, the author aims to migrate the tool into Eclipse Modelling Framework [171] to create an integrated toolset that could be used to create (or import) Sequence Diagrams, perform the transformation, conduct Petri Net analysis, and finally produce a report for the system designer. The tool development plan also includes a module for automated tool-based synthesis of Sequence Diagrams using the algorithms presented in this thesis; as well as the potential to add to the number of algorithms.

The synthesis techniques presented in this thesis adopted the top-down synthesis method to create a message refinement algorithm, and presented two specific cases where bottom-up synthesis method may be adopted in Sequence Diagrams. However, as discussed in Section 6.2, there are various other methods of synthesis in Petri Nets. Research is still in progress to determine which synthesis methods in Petri Nets could be adopted in Sequence Diagrams. Research is also being conducted to produce a generic way that all synthesis methods in Petri Net could be adopted seamlessly in Sequence Diagrams. One approach for seamless adoption of synthesis methods is through a bi-directional model transformation between Sequence Diagrams and Petri Nets where synthesis could be performed in Petri Nets and the results transformed back into Sequence Diagrams. However this could be complicated since the Petri Net language is more expressive that Sequence Diagrams.

Finally, the author is also in process of enhancing the Sequence Diagram metamodel used in the SD2PN model transformation to include OCL constraints. OCL is a text-based language that uses first-order logic statements to provide constraints of the model elements in UML. Translating these constraints into Petri Nets requires a higher level of Petri Nets that could analyse logical statements – Coloured Petri Nets [74]. The enhanced model transformation will preserve all the results of SD2PN (i.e. Free Choice Petri Nets, semantic preservation) but with an added capability to analyse logic as well as structure, behaviour and performance.

APPENDIX A

TRANSFORMING SEQUENCE DIAGRAM FRAGMENTS AND PETRI NET BLOCKS FROM SD2PN TRANSFORMATION RULES INTO LES

A.1 Sequence Diagram Fragments to LES

The translation of the Sequence Diagram fragments from the SD2PN transformation rules are based on the outline presented in Section 2.3.1 using the semantic mapping by [40].

Message

In Sequence Diagrams, the fragment message is described by two events; e_1 the event that describes the sending of the message, and e_2 that describes the receiving of the message. Since both events are causal, and belong to the same *scope* (this is true for any case since

messages are horizontal and there can never be a scenario that the sending and receiving events of a message exist under different *scopes*). This results in the LES in Figure 26.

Alternative

For the Sequence Diagram fragment *alternative,* the fragment has an initial location l_1 that represents the beginning of the fragment. Since the SD2PN transformation rules signifies two operands in the *alternative* fragment, there are 2 scopes; alt(2)#1 and alt(2)#2. Signifying the end of the *alternative* fragment is the location l_2. There is only one *alt_loc* for location l_1 since there are no choices or concurrencies; as such the event e_1 is the starting point of the LES. In location l_2 however, there are two possible *alt_loc* since the *alternative* fragment produces two different scenarios. As such, the location $l_2$produces two events $e_2$and e_3. Since e_2 and e_3 are conflicting events, where there are no sets of execution traces that contain both events; the symbol '#´ is placed in between the events denoting conflicting behaviour. After the addition of *placeholders* to represent the *placeholders* in Sequence Diagrams, the resulting LES is presented in Figure 26.

Option

The *option* fragment is semantically equivalent to the *alternative* fragment as discussed in Section 4.1.2.3. As such, the transformation is similar to the transformation of *alternative* fragments.

Break

The *break* fragment also has a similar construct to the *alternative* fragment, but with just one *placeholder*. It still consist of two locations l_1 and l_2 where l_1 has an *alt_loc* of 1 and l_2 has an *alt_loc* of 2 – generating two conflicting events $e_2$and e_3 as presented in Figure 26.

Parallel

In Sequence Diagrams, a *parallel* fragment has an initial location l_1. This location signifies the beginning of the fragment. Inside the fragment, there are 2 scopes; par(2)#1 and par(2)#2 as described in Section 2.3. These scopes represent the parallel events that occur inside the fragment. After the execution of these events, a location l_2 signifies the end of the fragment. Since both l_1 and l_2 has an *alt_loc* of 1, there is only 1 event to represent each these locations, e_1 and e_2 such that e_1 forks into the 2 scopes of events and merge into e_2. This creates an LES as shown in Figure 26.

A.2 Petri Net Blocks to LES

The translation of Petri Net blocks into LES makes use of the unfolding method presented in [41] as presented in Section 2.3.2.

Message

The unfolding of the Petri Net block representing message is straight-forward where there are two states in the Petri Net in form of places s_1 and s_2. The causal relationship between the places ensures the LES as presented in Figure 26.

Alternative

The Petri Net block that represents the fragment *alternative* begins with a state represented by the place s_1, creating e_1 in the LES. However, 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 presented in Figure 26.

Option

This unfolding is similar to *alternative*.

Break

This unfolding is similar to *alternative* but with one *placeholder*.

Parallel

In the block of Petri Net representing *parallel*, it starts with a place s_1 and ends with a place s_2. They can be represented as events e_1 and e_2 respectively with e_1 forking out into the *placeholders* and merging at e_2. This results in the LES representation in Figure 26.

APPENDIX B

SOURCE CODE FOR SD2PN TRANSFORMER

```
package csv.parser;

import sequencediagram.*;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import javax.swing.JFileChooser;
import javax.swing.filechooser.FileFilter;

public class CSVParser {

    private JFileChooser fileChooser;
    private File[] files;
    private int status = JFileChooser.CANCEL_OPTION;
    private ArrayList<String[]> data;

    public CSVParser() {

        data = new ArrayList<String[]>();
        //("D:/Summer Project/Ariff");
        fileChooser = new JFileChooser("C:/Documents and Settings/Ben Sab/My
Documents/University of Birmingham/MSc Advanced Computer Science/Summer
Project/Ariff");
        fileChooser.setAcceptAllFileFilterUsed(false);
```

```
        FileFilter filter1 = new ExtensionFileFilter("CSV files", "CSV");
        fileChooser.setFileFilter(filter1);

        fileChooser.setDialogTitle("Open CSV files");
        fileChooser.setMultiSelectionEnabled(true);

        status = fileChooser.showOpenDialog(null);

        if (status == JFileChooser.APPROVE_OPTION) {
            files = fileChooser.getSelectedFiles();
        } else {
            System.exit(0);
        }

        parse();

    }

    public final File[] getFiles() {
        return files;
    }

    public final boolean isFileSelected(String fileName) {

        for (int i = 0; i < getFiles().length; i++) {
            if (getFiles()[i].getName().contains(fileName)) {
                return true;
            }
        }

        return false;

    }

    private final void parse() {

        if (files.length != 0) {

            try {

                for (int i = 0; i < files.length; i++) {

                    BufferedReader     reader     =     new     BufferedReader(new
FileReader(files[i]));

                    data.add(new String[]{files[i].getName()});

                    String line;
                    String[] columns;

                    while ((line = reader.readLine()) != null) {
                        if (line.contains("[")) {

                            Pattern pattern = Pattern.compile("\\[[^\\]]*\\]");
                            Matcher matcher = pattern.matcher(line);

                            while (matcher.find()) {
                                line      =      line.replaceFirst("\\[[^\\]]*\\]",
matcher.group().substring(1, matcher.group().length() - 1).replace(",", ";"));
                            }

                        }
                        line += " ";
                        columns = line.split(",");
                        data.add(columns);
```

```
                }

            }

        } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
        }

    }

}

public final ArrayList<String[]> getAllData() {
    return data;
}

public final ArrayList<String> getColumnInFile(String fileName, int
columnNumber) {

    ArrayList<String> result = new ArrayList<String>();

    int err = 0;

    if (files.length != 0 && data.size() > 0) {

        try {

            for (int i = 0; i < data.size(); i++) {
                if (data.get(i)[0].contains(fileName)) {
                    for (int j = i + 1; j < data.size(); j++) {
                        if (data.get(j)[0].endsWith(".csv")) {
                            break;
                        }
                        result.add(data.get(j)[columnNumber]);
                    }
                    break;
                }
            }

            if (result.size() == 0) {
                System.err.println("The file: '" + fileName + "' was not
imported.");
                err = -1;
            }

            return result;

        } catch (ArrayIndexOutOfBoundsException aioobe) {
            System.err.println("Column number does not exist.");
            err = -1;
        }

    }

    if (err == 0 && result.size() == 0) {
        System.err.println("The file: '" + fileName + "' is either empty or no
files have been imported.");
    }

    return result;

}

public final ArrayList<String> getColumnInFile(String fileName, String
columnName) {
```

```
        ArrayList<String> result = new ArrayList<String>();

        int err = 0;

        if (files.length != 0 && data.size() > 0) {

            try {

                for (int i = 0; i < data.size(); i++) {
                    if (data.get(i)[0].contains(fileName)) {
                        int columnNumber = -1;
                        for (int j = i + 1; j < data.size(); j++) {
                            if (j == i + 1) {
                                for (int z = 0; z < data.get(j).length; z++) {
                                    if (data.get(j)[z].trim().equals(columnName)) {
                                        columnNumber = z;
                                        break;
                                    }
                                }
                                if (columnNumber == -1) {
                                    System.err.println("The column: '" + columnName
+ "' does not exist in the file: '" + fileName + "'.");
                                    err = -1;
                                    break;
                                }
                                j++;
                            }
                            if (data.get(j)[0].endsWith(".csv")) {
                                break;
                            }
                            result.add(data.get(j)[columnNumber]);
                        }
                        break;
                    }
                }

                if (err == 0 && result.size() == 0) {
                    System.err.println("The  file:  '"  +  fileName  +  "'  was  not
imported.");
                    err = -1;
                }

                return result;

            } catch (ArrayIndexOutOfBoundsException aioobe) {
                System.err.println("Column number does not exist.");
                err = -1;
            }

        }

        if (err == 0 && result.size() == 0) {
            System.err.println("The file: '" + fileName + "' is either empty or no
files have been imported.");
        }

        return result;

    }

    public final ArrayList<String[]> getFileData(String fileName) {

        ArrayList<String[]> result = new ArrayList<String[]>();
```

```
        if (files.length != 0 && data.size() > 0) {

            for (int i = 0; i < data.size(); i++) {
                if (data.get(i)[0].contains(fileName)) {
                    for (int j = i + 1; j < data.size(); j++) {
                        if (data.get(j)[0].endsWith(".csv")) {
                            break;
                        }
                        result.add(data.get(j));
                    }
                    break;
                }
            }

            if (result.size() == 0) {
                System.err.println("The file: '" + fileName + "' was not
imported.");
            }

            return result;

        }

        System.err.println("The file: '" + fileName + "' is empty or no files have
been imported.");

        return result;
    }

    public final ArrayList<EventOccurrence> getEventOccurences() {
        ArrayList<EventOccurrence> events = new ArrayList<EventOccurrence>();

        if (isFileSelected("_occurrencespec.csv")) {

            ArrayList<String> id = getColumnInFile("_occurrencespec.csv", "id");
            ArrayList<String> context = getColumnInFile("_occurrencespec.csv",
"context");

            for (int i = 0; i < id.size(); i++) {
                EventOccurrence event = new EventOccurrence(id.get(i),
context.get(i), i);
                events.add(event);
            }

        } else {
            System.err.println("The following file was not imported:
'_occurrencespec.csv'.");
        }

        return events;
    }

    public final ArrayList<CombinedFragments> getCombinedFragments() {
        ArrayList<CombinedFragments> combinedFragments = new
ArrayList<CombinedFragments>();

        if (isFileSelected("_combinedfragment.csv")) {

            ArrayList<String> id = getColumnInFile("_combinedfragment.csv", "id");
            ArrayList<String> context = getColumnInFile("_combinedfragment.csv",
"context");
            ArrayList<String> operator = getColumnInFile("_combinedfragment.csv",
"operator");
            ArrayList<String> operands = getColumnInFile("_combinedfragment.csv",
"operands");
```

```
            for (int i = 0; i < id.size(); i++) {
                int op = -1;
                if (operator.get(i).equals("alt")) {
                    op = InteractionOperatorKind.ALT;
                } else if (operator.get(i).equals("break")) {
                    op = InteractionOperatorKind.BREAK;
                } else if (operator.get(i).equals("opt")) {
                    op = InteractionOperatorKind.OPT;
                } else if (operator.get(i).equals("par")) {
                    op = InteractionOperatorKind.PAR;
                }
                CombinedFragments          combinedFragment          =          new
CombinedFragments(id.get(i), context.get(i), op, operands.get(i));
                combinedFragments.add(combinedFragment);
            }

        } else {
            System.err.println("The    following    file    was    not    imported:
'_combinedfragment.csv'.");
        }

        return combinedFragments;
    }

    public final ArrayList<Lifeline> getLifeline() {
        ArrayList<Lifeline> lifelines = new ArrayList<Lifeline>();

        if (isFileSelected("_lifeline.csv")) {

            ArrayList<String> id = getColumnInFile("_lifeline.csv", "id");
            ArrayList<String> name = getColumnInFile("_lifeline.csv", "name");
            ArrayList<String>     context     =     getColumnInFile("_lifeline.csv",
"context");

            for (int i = 0; i < id.size(); i++) {
                Lifeline   lifeline   =   new   Lifeline(id.get(i),   name.get(i),
context.get(i));
                lifelines.add(lifeline);
            }

        } else {
            System.err.println("The    following    file    was    not    imported:
'_lifeline.csv'.");
        }

        return lifelines;
    }

    public  final  ArrayList<Message>  getMessages(ArrayList<EventOccurrence>  events)
{

        ArrayList<Message> messages = new ArrayList<Message>();

        if                    (isFileSelected("_message.csv")                    &&
isFileSelected("_occurrencespec.csv")) {
            ArrayList<String> id = getColumnInFile("_message.csv", "id");
            ArrayList<String> label = getColumnInFile("_message.csv", "name");
            ArrayList<String> context = getColumnInFile("_message.csv", "context");
            ArrayList<String>     sendEvent     =     getColumnInFile("_message.csv",
"sendevent");
            ArrayList<String>    receiveEvent    =    getColumnInFile("_message.csv",
"receiveevent");
            for (int i = 0; i < id.size(); i++) {
                EventOccurrence send = null;
```

```
                EventOccurrence receive = null;

                boolean sendFound = false;
                boolean receiveFound = false;
                for (int j = 0; j < events.size(); j++) {
                    if (sendFound && receiveFound) {
                        break;
                    }
                    if (sendEvent.get(i).equals(events.get(j).getID())) {
                        send = events.get(j);
                        sendFound = true;
                    } else if (receiveEvent.get(i).equals(events.get(j).getID())) {
                        receive = events.get(j);
                        receiveFound = true;
                    }
                }
                if (send == null || receive == null) {
                    // There are one or more errors in the files
                    System.err.println("getMessages: There are one or more errors
in the files - events not found");
                } else {
                    Message msg = new Message(id.get(i), label.get(i),
context.get(i), send, receive);
                    messages.add(msg);
                }
            }
        } else {
            System.err.println("Some of the following files were not imported:
'_occurrencespec.csv', '_message.csv'.");
        }

        return messages;

    }

    public static void main(String[] args) {

        CSVParser parser = new CSVParser();
        ArrayList<EventOccurrence> events = parser.getEventOccurences();
        ArrayList<Message> messages = parser.getMessages(events);
        ArrayList<CombinedFragments> combinedfrags = parser.getCombinedFragments();
        ArrayList<Lifeline> lifelines = parser.getLifeline();

        for (int i = 0; i < messages.size(); i++) {
            System.out.println("Message name: " + messages.get(i).getLabel());
            System.out.println("Message ID: " + messages.get(i).getID());
            System.out.println("                    Send-Event     ID:     "     +
messages.get(i).getSendEvent().getID());
            System.out.println("                  Receive-Event     ID:     "     +
messages.get(i).getReceiveEvent().getID());
        }

        System.out.println("-----------------------");
        for (int i = 0; i < lifelines.size(); i++) {
            System.out.println("Lifeline name: " + lifelines.get(i).getName());
            System.out.println("Lifeline ID: " + lifelines.get(i).getID());
        }

        System.out.println("-----------------------");
        for (int i = 0; i < combinedfrags.size(); i++) {
            System.out.println("CombinedFragment          ID:          "          +
combinedfrags.get(i).getID());
            System.out.println("CombinedFragment    No.    of    Fragments:    "    +
combinedfrags.get(i).getNumberOfFragments());
        }
```

```
        System.out.println("-----------------------");
        Interaction  interaction  =  new  Interaction(lifelines,  messages,  events,
combinedfrags);

        for (int i = 0; i < interaction.getEventOccurrences().size(); i++) {
            for (int j = 0; j < interaction.getMessages().size(); j++) {
                if
(interaction.getMessage(j).getReceiveEvent().getID().equals(interaction.getEventOcc
urrence(i).getID())) {
                    System.out.println("Receiver:                 "                +
interaction.getMessage(j).getLabel());
                }                                  else                                if
(interaction.getMessage(j).getSendEvent().getID().equals(interaction.getEventOccurr
ence(i).getID())) {
                    System.out.println("Sender:                  "                 +
interaction.getMessage(j).getLabel());
                }
            }
        }
    }

    class ExtensionFileFilter extends FileFilter {

        String description;
        String extensions[];

        public ExtensionFileFilter(String description, String extension) {
            this(description, new String[]{extension});
        }

        public ExtensionFileFilter(String description, String extensions[]) {
            if (description == null) {
                this.description = extensions[0];
            } else {
                this.description = description;
            }
            this.extensions = (String[]) extensions.clone();
            toLower(this.extensions);
        }

        private void toLower(String array[]) {
            for (int i = 0, n = array.length; i < n; i++) {
                array[i] = array[i].toLowerCase();
            }
        }

        public String getDescription() {
            return description;
        }

        public boolean accept(File file) {
            if (file.isDirectory()) {
                return true;
            } else {
                String path = file.getAbsolutePath().toLowerCase();
                for (int i = 0, n = extensions.length; i < n; i++) {
                    String extension = extensions[i];
                    if  ((path.endsWith(extension)  &&  (path.charAt(path.length()  -
extension.length() - 1)) == '.')) {
                        return true;
                    }
                }
            }
            return false;
```

```
        }
    }
}

package csv.parser;

import java.util.ArrayList;
import javax.swing.JFrame;
import petrinet.*;
import sequencediagram.*;
import sitra.*;

public class SD2PN {

    private Transformer transformer;

    public SD2PN(Transformer transformer) {
        this.transformer = transformer;
    }

    public ArrayList<PetriNet> createPetriNets(Interaction interaction) {
        ArrayList<PetriNet> petriArray = new ArrayList<PetriNet>();

        for (int j = 0; j < interaction.getEventOccurrences().size(); j++) {
            for (int i = 0; i < interaction.getMessages().size(); i++) {
                if
(interaction.getEventOccurrence(j).getID().equals(interaction.getMessage(i).getSend
Event().getID()) ||

interaction.getEventOccurrence(j).getID().equals(interaction.getMessage(i).getRecei
veEvent().getID())) {
                    if (!interaction.getMessage(i).getLabel().trim().equals("")) {
                        petriArray.add((PetriNet)
transformer.transform(interaction.getMessage(i)));
                    }
                    j++;
                }
            }
        }

        for (int i = 0; i < interaction.getCombinedFragments().size(); i++) {
            petriArray.add((PetriNet)
transformer.transform(interaction.getCombinedFragment(i)));
        }

        // the next line should be modified because the first message might be in a
placeholder
        petriArray.get(0).getPlace(0).setMark(new Mark(1));

        return petriArray;
    }

    public  PetriNet  morphAndSubstitute(ArrayList<PetriNet>  petrinets,  Interaction
interaction) {

        // STEP 1
        for (int i = 1; i < petrinets.size(); i++) {
            if (petrinets.get(i).getName().trim().equals("") &&
                    petrinets.get(i - 1).getName().trim().equals("") &&
                    petrinets.get(i).getContext().trim().equals(petrinets.get(i    -
1).getContext().trim())) {

                morph(petrinets.get(i - 1), petrinets.get(i));
                petrinets.remove(i - 1);
                i--;
```

```
            }
        }

        // STEP 2
        for (int i = 0; i < petrinets.size(); i++) {
            if (petrinets.get(i).hasPlaceHolder()) {
                for (int j = 0; j < petrinets.get(i).getPlaceHolders().size(); j++)
{
                    // in the next line the condition (k < i) is assuming that
                    //  the  messages  are  added  first  and  then  the  combined
fragments.
                    for (int k = 0; k < i; k++) {
                        if (petrinets.get(k).getName().trim().equals("") &&

petrinets.get(i).getPlaceHolder(j).getID().trim().equals(petrinets.get(k).getContex
t().trim())) {
                            substitute(petrinets.get(i), petrinets.get(k));
                            petrinets.remove(petrinets.get(k));
                            i--;
                            j--;
                            break;
                        }
                    }
                }
            }
        }

        // STEP 3
        // this case has yet to be tested
        for (int i = 0; i < petrinets.size(); i++) {
            for (int j = 0; j < petrinets.size(); j++) {
                if    (petrinets.get(i).hasPlaceHolder()    &&    i    !=    j    &&
!petrinets.get(i).getName().trim().equals("")                                    &&
!petrinets.get(j).getName().trim().equals("")) {
                    if
(petrinets.get(i).getID().trim().equals(petrinets.get(j).getContext().trim())) {
                        substitute(petrinets.get(i), petrinets.get(j));
                        petrinets.remove(petrinets.get(j));
                        i = 0;
                        break;
                    }
                }
            }
        }

        // STEP 4
        for (int i = 0; i < petrinets.size(); i++) {
            if (i + 1 < petrinets.size()) {
                morph(petrinets.get(i), petrinets.get(i + 1));
                petrinets.remove(i);
                i--;
            }
        }

        return petrinets.get(0);

    }

    public PetriNet morph(PetriNet pn1, PetriNet pn2) {
        boolean noErr = false;
        Place last = pn1.getLastPlace();
        ArrayList<Arc> all = pn2.getAllArcs(pn2.getFirstPlace());
        for (int i = 0; i < all.size(); i++) {
            all.get(i).setPlace(last);
            noErr = true;
```

```
        }
        if (noErr) {
            pn2.removePlace(pn2.getPlace(0));
            pn2.addArcs(pn1.getArcs());
            pn2.addPlaces(pn1.getPlaces());
            pn2.addTransitions(pn1.getTransitions());
            pn2.addPlaceHolders(pn1.getPlaceHolders());
        }
        return pn2;
    }

    public PetriNet substitute(PetriNet cf, PetriNet pn) {
        if (cf != null && pn != null) {
            if (cf.hasPlaceHolder()) {
                for (int i = 0; i < cf.getPlaceHolders().size(); i++) {
                    if
(cf.getPlaceHolder(i).getID().trim().equals(pn.getContext().trim())) {
                        boolean in = false;
                        boolean out = false;
                        for (int j = 0; j < cf.getArcs().size(); j++) {
                            if          (cf.getArc(j).hasPlaceHolder()          &&
cf.getArc(j).getPlaceHolder().equals(cf.getPlaceHolder(i))) {
                                if          (cf.getArc(j).getDirection()         ==
Arc.TRANSITION_TO_PLACEHOLDER) {
                                    cf.getArc(j).setPlace(pn.getPlace(0));

cf.getArc(j).setDirection(Arc.TRANSITION_TO_PLACE);
                                    in = true;
                                } else {

cf.getArc(j).setPlace(pn.getPlace(pn.getPlaces().size() - 1));

cf.getArc(j).setDirection(Arc.PLACE_TO_TRANSITION);
                                    out = true;
                                }
                            }
                            if (in && out) {
                                cf.removePlaceHolder(cf.getPlaceHolder(i));
                                cf.removePlace(cf.getArc(j).getPlace());
                                cf.addArcs(pn.getArcs());
                                cf.addPlaces(pn.getPlaces());
                                cf.addTransitions(pn.getTransitions());
                                cf.addPlaceHolders(pn.getPlaceHolders());
                                return cf;
                            }
                        }
                        break;
                    }
                }
                return null;
            } else {
                System.err.println("SD2PN: SUBSTITUTE: No PlaceHolder in the given
PetriNet.");
                return null;
            }
        } else {
            System.err.println("SD2PN: SUBSTITUTE: NullPointerException.");
            return null;
        }
    }

    public static void main(String[] args) {

        CSVParser parser = new CSVParser();
        ArrayList<EventOccurrence> events = parser.getEventOccurences();
```

```
        ArrayList<Message> messages = parser.getMessages(events);
        ArrayList<CombinedFragments> combinedfrags = parser.getCombinedFragments();
        ArrayList<Lifeline> lifelines = parser.getLifeline();
        Interaction interaction = new Interaction(lifelines, messages, events,
combinedfrags);

        ArrayList<Class<? extends Rule>> rules = new ArrayList<Class<? extends
Rule>>();
        rules.add(Rule1.class);
        rules.add(Rule2.class);
        rules.add(Rule3.class);
        rules.add(Rule4.class);
        rules.add(Rule5.class);

        SD2PN sd2pn = new SD2PN(new SimpleTransformerImpl(rules));

        PetriNet                                petrinet                                =
sd2pn.morphAndSubstitute(sd2pn.createPetriNets(interaction), interaction);

        for (int j = 0; j < petrinet.getArcs().size(); j++) {
            System.out.println("Place:                                  "                       +
petrinet.getArc(j).getPlace().toString().substring(petrinet.getArc(j).getPlace().to
String().indexOf("@")    +    1)    +    (petrinet.getArc(j).getDirection()    ==
Arc.PLACE_TO_TRANSITION  ?  "  --->  "  :  "  <---  ")  +  "Transition:  "  +
petrinet.getArc(j).getTransition().getName());
        }

    }
}

package petrinet;

public class Arc {

    public static final int TRANSITION_TO_PLACE = 0;
    public static final int PLACE_TO_TRANSITION = 1;
    public static final int PLACEHOLDER_TO_TRANSITION = 2;
    public static final int TRANSITION_TO_PLACEHOLDER = 3;
    private Place place;
    private PlaceHolder placeHolder;
    private Transition transition;
    private int direction;

    public Arc(Place place, Transition transition, int direction) {
        this.place = place;
        this.transition = transition;
        this.direction = direction;
        this.placeHolder = null;
    }

    public Arc(PlaceHolder placeHolder, Transition transition, int direction) {
        this.placeHolder = placeHolder;
        this.transition = transition;
        this.direction = direction;
        this.place = null;
    }

    public boolean hasPlaceHolder() {
        return (placeHolder != null);
    }

    public Place getPlace() {
        return place;
    }
```

```
    public PlaceHolder getPlaceHolder() {
        return placeHolder;
    }

    public Transition getTransition() {
        return transition;
    }

    public int getDirection() {
        return direction;
    }

    public void removePlaceHolder() {
        placeHolder = null;
    }

    public void setPlace(Place newPlace) {
        this.place = newPlace;
        removePlaceHolder();
    }

    public void setTransition(Transition newTransition) {
        this.transition = newTransition;
    }

    public void setDirection(int newDirection) {
        this.direction = newDirection;
    }
}

package petrinet;

public class Mark {

    private int numberOfTokens;

    public Mark(int numberOfTokens) {
        this.numberOfTokens = numberOfTokens;
    }

    public void removeTokens(int number) {
        if (numberOfTokens >= number) {
            numberOfTokens -= number;
        }
    }

    public void addTokens(int number) {
        numberOfTokens += number;
    }

    public void clearTokens() {
        numberOfTokens = 0;
    }
}

package petrinet;

import java.util.ArrayList;

public class PetriNet {

    private ArrayList<Place> places;
    private ArrayList<PlaceHolder> placeHolders;
    private ArrayList<Transition> transitions;
    private ArrayList<Arc> arcs;
```

```
    private ArrayList<Marking> markings;
    private String context;
    private String id;
    private String name;

    public PetriNet(String fragmentContext, String fragmentID, String fragmentName)
{
        places = new ArrayList<Place>();
        placeHolders = new ArrayList<PlaceHolder>();
        transitions = new ArrayList<Transition>();
        arcs = new ArrayList<Arc>();
        markings = new ArrayList<Marking>();
        context = fragmentContext;
        id = fragmentID;
        name = fragmentName;
    }

    public boolean addPlace(Place newPlace) {
        if (newPlace != null) {
            for (int i = 0; i < places.size(); i++) {
                if (newPlace.equals(places.get(i))) {
                    return false;
                }
            }
            places.add(newPlace);
            return true;
        }
        return false;
    }

    public void addPlaces(ArrayList<Place> newPlaces) {
        for (int i = 0; i < newPlaces.size(); i++) {
            addPlace(newPlaces.get(i));
        }
    }

    public boolean addPlaceHolder(PlaceHolder newPlaceHolder) {
        if (newPlaceHolder != null) {
            for (int i = 0; i < placeHolders.size(); i++) {
                if (newPlaceHolder.equals(placeHolders.get(i))) {
                    return false;
                }
            }
            placeHolders.add(newPlaceHolder);
            return true;
        }
        return false;
    }

    public void addPlaceHolders(ArrayList<PlaceHolder> newPlaceHolders) {
        for (int i = 0; i < newPlaceHolders.size(); i++) {
            addPlaceHolder(newPlaceHolders.get(i));
        }
    }

    public boolean addTransition(Transition newTransition) {
        if (newTransition != null) {
            for (int i = 0; i < transitions.size(); i++) {
                if (newTransition.getName().equals(transitions.get(i).getName())) {
                    return false;
                }
            }
            transitions.add(newTransition);
            return true;
        }
```

```
        return false;
    }

    public void addTransitions(ArrayList<Transition> newTransitions) {
        for (int i = 0; i < newTransitions.size(); i++) {
            addTransition(newTransitions.get(i));
        }
    }

    public void addArc(Arc newArc) {
        arcs.add(newArc);
        if (!newArc.hasPlaceHolder()) {
            addPlace(arcs.get(arcs.size() - 1).getPlace());
        } else {
            addPlaceHolder(arcs.get(arcs.size() - 1).getPlaceHolder());
        }
        addTransition(arcs.get(arcs.size() - 1).getTransition());
    }

    public void addArcs(ArrayList<Arc> newArcs) {
        for (int i = 0; i < newArcs.size(); i++) {
            addArc(newArcs.get(i));
        }
    }

    public void addMarking(Marking newMarking) {
        markings.add(newMarking);
    }

    /** This method has a meaning if the petrinet has only one starting place and
one ending place */
    public Place getFirstPlace() {
        for (int i = 0; i < places.size(); i++) {
            ArrayList<Arc> input = getInputArcs(places.get(i));
            if (input.size() == 0) {
                return places.get(i);
            }
        }
        return null;
    }

    /** This method has a meaning if the petrinet has only one starting place and
one ending place */
    public Place getLastPlace() {
        for (int i = 0; i < places.size(); i++) {
            ArrayList<Arc> output = getOutputArcs(places.get(i));
            if (output.size() == 0) {
                return places.get(i);
            }
        }
        return null;
    }

    public ArrayList<Arc> getAllArcs(Place place) {
        ArrayList<Arc> all = new ArrayList<Arc>();

        for (int i = 0; i < arcs.size(); i++) {
            if (arcs.get(i).getPlace().equals(place)) {
                all.add(arcs.get(i));
            }
        }

        return all;
    }
```

```
    public ArrayList<Arc> getInputArcs(Place place) {
        ArrayList<Arc> all = new ArrayList<Arc>();

        for (int i = 0; i < arcs.size(); i++) {
            if (arcs.get(i).getPlace().equals(place) && arcs.get(i).getDirection()
== Arc.TRANSITION_TO_PLACE) {
                all.add(arcs.get(i));
            }
        }

        return all;
    }

    public ArrayList<Arc> getOutputArcs(Place place) {
        ArrayList<Arc> all = new ArrayList<Arc>();

        for (int i = 0; i < arcs.size(); i++) {
            if (arcs.get(i).getPlace().equals(place) && arcs.get(i).getDirection()
== Arc.PLACE_TO_TRANSITION) {
                all.add(arcs.get(i));
            }
        }

        return all;
    }

    public ArrayList<Arc> getAllArcs(Transition t) {
        ArrayList<Arc> all = new ArrayList<Arc>();

        for (int i = 0; i < arcs.size(); i++) {
            if (arcs.get(i).getTransition().equals(t)) {
                all.add(arcs.get(i));
            }
        }

        return all;
    }

    public ArrayList<Arc> getInputArcs(Transition t) {
        ArrayList<Arc> all = new ArrayList<Arc>();

        for (int i = 0; i < arcs.size(); i++) {
            if (arcs.get(i).getTransition().equals(t) &&
(arcs.get(i).getDirection() == Arc.PLACE_TO_TRANSITION ||
arcs.get(i).getDirection() == Arc.PLACEHOLDER_TO_TRANSITION)) {
                all.add(arcs.get(i));
            }
        }

        return all;
    }

    public ArrayList<Arc> getOutputArcs(Transition t) {
        ArrayList<Arc> all = new ArrayList<Arc>();

        for (int i = 0; i < arcs.size(); i++) {
            if (arcs.get(i).getTransition().equals(t) &&
(arcs.get(i).getDirection() == Arc.TRANSITION_TO_PLACE ||
arcs.get(i).getDirection() == Arc.TRANSITION_TO_PLACEHOLDER)) {
                all.add(arcs.get(i));
            }
        }

        return all;
    }
```

```
public Place getPlace(int index) {
    return places.get(index);
}

public PlaceHolder getPlaceHolder(int index) {
    return placeHolders.get(index);
}

public Transition getTransition(int index) {
    return transitions.get(index);
}

public Arc getArc(int index) {
    return arcs.get(index);
}

public Marking getMarking(int index) {
    return markings.get(index);
}

public ArrayList<Place> getPlaces() {
    return places;
}

public ArrayList<PlaceHolder> getPlaceHolders() {
    return placeHolders;
}

public ArrayList<Transition> getTransitions() {
    return transitions;
}

public ArrayList<Arc> getArcs() {
    return arcs;
}

public ArrayList<Marking> getMarkings() {
    return markings;
}

public String getContext() {
    return context;
}

public String getID() {
    return id;
}

public String getName() {
    return name;
}

public boolean hasPlaceHolder() {
    return !placeHolders.isEmpty();
}

public void setContext(String context) {
    this.context = context;
}

public void setID(String id) {
    this.id = id;
}
```

```
    public void setName(String name) {
        this.name = name;
    }

    public void removePlace(Place place) {
        places.remove(place);
    }

    public void removePlaceHolder(PlaceHolder placeHolder) {
        placeHolders.remove(placeHolder);
    }

    public void removeTransitions(Transition transition) {
        transitions.remove(transition);
    }

    public void removeArc(Arc arc) {
        arcs.remove(arc);
    }

    public void removeMarking(Marking marking) {
        markings.remove(marking);
    }
}

package petrinet;

public class Place {

    private Mark mark;

    public Place(Mark mark) {
        this.mark = mark;
    }

    public Mark getMark() {
        return mark;
    }

    public void setMark(Mark newMark) {
        mark = newMark;
    }
}

package petrinet;

public class PlaceHolder extends PetriNet{

    public   PlaceHolder(String   fragmentContext,   String   fragmentID,   String
fragmentName){
        super(fragmentContext, fragmentID, fragmentName);
    }

}

package petrinet;

public class Transition {

    private String name;

    public Transition(String name) {
        this.name = name;
    }
```

```
    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }
}

package sequencediagram;

public class CombinedFragments {

    private String id;
    private String context;
    private int operatorKind;
    private String[] fragments;
    private int numberOfFragments;

    public CombinedFragments(String id, String context, int operatorKind, String
fragments) {
        this.id = id;
        this.context = context;
        this.operatorKind = operatorKind;
        this.fragments = fragments.split(";");
        this.numberOfFragments = this.fragments.length;
    }

    public String getID() {
        return id;
    }

    public String getContext() {
        return context;
    }

    public int getOperator() {
        return operatorKind;
    }

    public String[] getFragments(){
        return fragments;
    }

    public int getNumberOfFragments(){
        return numberOfFragments;
    }
}

package sequencediagram;

public class EventOccurrence {

    private String id;
    private String context;
    private int index;
    // we need to have an index here for ordering
    public EventOccurrence(String id, String context, int index) {
        this.id = id;
        this.context = context;
        this.index = index;
    }

    public String getID() {
        return id;
```

```
    }

    public void setID(String id) {
        this.id = id;
    }

    public String getContext() {
        return context;
    }

    public void setContext(String context) {
        this.context = context;
    }

    public int getIndex() {
        return index;
    }

    public void setIndex(int index) {
        this.index = index;
    }
}

package sequencediagram;

import java.util.ArrayList;

public class GeneralOrdering {

    public GeneralOrdering() {
    }

    public    EventOccurrence    getEventAfter(ArrayList<EventOccurrence>    events,
EventOccurrence event) {
        for (int i = 0; i < events.size() - 1; i++) {
            if (events.get(i).equals(event)) {
                return events.get(i + 1);
            }
        }
        return null;
    }

    public    EventOccurrence    getEventBefore(ArrayList<EventOccurrence>    events,
EventOccurrence event) {
        for (int i = 1; i < events.size(); i++) {
            if (events.get(i).equals(event)) {
                return events.get(i - 1);
            }
        }
        return null;
    }

    public EventOccurrence getEvent(ArrayList<EventOccurrence> events, String id) {
        for (int i = 0; i < events.size(); i++) {
            if (events.get(i).getID().equals(id)) {
                return events.get(i);
            }
        }
        return null;
    }
}

package sequencediagram;

import java.util.ArrayList;
```

```
public class Interaction {

    public ArrayList<Lifeline> lifelines;
    public ArrayList<Message> messages;
    public ArrayList<EventOccurrence> events;
    public ArrayList<CombinedFragments> combinedFragments;
    public GeneralOrdering generalOrdering;

    public Interaction(final ArrayList<Lifeline> lifelines, final
ArrayList<Message> messages, final ArrayList<EventOccurrence> events, final
ArrayList<CombinedFragments> combinedFragments) {
        this.lifelines = new ArrayList<Lifeline>(lifelines);
        this.messages = new ArrayList<Message>(messages);
        this.combinedFragments = new
ArrayList<CombinedFragments>(combinedFragments);
        this.events = new ArrayList<EventOccurrence>(events);
        this.generalOrdering = new GeneralOrdering();
    }

    public Message getMessage(String id) {
        for (int i = 0; i < messages.size(); i++) {
            if (messages.get(i).getID().equals(id)) {
                return messages.get(i);
            }
        }
        return null;
    }

    public Lifeline getLifeline(String id) {
        for (int i = 0; i < lifelines.size(); i++) {
            if (lifelines.get(i).getID().equals(id)) {
                return lifelines.get(i);
            }
        }
        return null;
    }

    public EventOccurrence getEventOccurrence(String id) {
        for (int i = 0; i < events.size(); i++) {
            if (events.get(i).getID().equals(id)) {
                return events.get(i);
            }
        }
        return null;
    }

    public CombinedFragments getCombinedFragment(String id) {
        for (int i = 0; i < combinedFragments.size(); i++) {
            if (combinedFragments.get(i).getID().equals(id)) {
                return combinedFragments.get(i);
            }
        }
        return null;
    }

    public Message getMessage(int index) {
        return messages.get(index);
    }

    public Lifeline getLifeline(int index) {
        return lifelines.get(index);
    }

    public EventOccurrence getEventOccurrence(int index) {
```

```
        return events.get(index);
    }

    public CombinedFragments getCombinedFragment(int index) {
        return combinedFragments.get(index);
    }

    public ArrayList<Message> getMessages() {
        return messages;
    }

    public ArrayList<Lifeline> getLifelines() {
        return lifelines;
    }

    public ArrayList<EventOccurrence> getEventOccurrences() {
        return events;
    }

    public ArrayList<CombinedFragments> getCombinedFragments() {
        return combinedFragments;
    }

    public GeneralOrdering getGeneralOrdering() {
        return generalOrdering;
    }
}

package sequencediagram;

public class InteractionOperand {

//    public InteractionConstraint theInteractionConstraint;
    private String id;

    public InteractionOperand(String id) {
        this.id= id;
    }

    public String getID() {
        return id;
    }

    public void setID(String id) {
        this.id = id;
    }
}

package sequencediagram;

public class InteractionOperatorKind {

    public final static int ALT = 0;
    public final static int OPT = 1;
    public final static int BREAK = 2;
    public final static int PAR = 3;
}

package sequencediagram;

public class Lifeline {

    private String id;
    private String name;
    private String context;
```

```
    public Lifeline(String id, String name, String context) {
        this.id = id;
        this.name = name;
        this.context = context;
    }

    public final String getID() {
        return id;
    }

    public final String getName() {
        return name;
    }

    public final String getContext() {
        return context;
    }
}

package sequencediagram;

public class Message {

    private String id;
    private String label;
    private String context;
    private EventOccurrence sendEvent;
    private EventOccurrence receiveEvent;

    public  Message(String  id,  String  label,  String  context,  EventOccurrence
sendEvent, EventOccurrence receiveEvent) {
        this.id = id;
        this.label = label;
        this.context = context;
        this.sendEvent = sendEvent;
        this.receiveEvent = receiveEvent;
    }

    public final String getID() {
        return id;
    }

    public final String getLabel() {
        return label;
    }

    public final String getContext() {
        return context;
    }

    public final EventOccurrence getSendEvent(){
        return sendEvent;
    }

    public final EventOccurrence getReceiveEvent(){
        return receiveEvent;
    }

    public final void setSendEvent(EventOccurrence eo){
        sendEvent = eo;
    }

    public final void setReceiveEvent(EventOccurrence eo){
        receiveEvent = eo;
```

```
    }

}

package sitra;

public interface Rule<S,T> {
       boolean check(S source);
       T build(S source, Transformer t);
       void setProperties(T target, S source, Transformer t);
}

package sitra;

import petrinet.Arc;
import petrinet.Mark;
import petrinet.PetriNet;
import petrinet.Place;
import petrinet.Transition;
import sequencediagram.Message;

public class Rule1 implements Rule {

    public Rule1() {
    }

    public boolean check(Object source) {
        return source instanceof Message;
    }

    public PetriNet build(Object source, Transformer t) {
        PetriNet pn = new PetriNet(((Message) source).getSendEvent().getContext(),
((Message) source).getID(), "");
        Transition tran = new Transition(((Message) source).getLabel());
        pn.addArc(new Arc(new Place(new Mark(0)), tran, Arc.PLACE_TO_TRANSITION));
        pn.addArc(new Arc(new Place(new Mark(0)), tran, Arc.TRANSITION_TO_PLACE));
        return pn;
    }

    public void setProperties(Object target, Object source, Transformer t) {
    }
}

package sitra;

import petrinet.Arc;
import petrinet.Mark;
import petrinet.PetriNet;
import petrinet.Place;
import petrinet.PlaceHolder;
import petrinet.Transition;
import sequencediagram.CombinedFragments;
import sequencediagram.InteractionOperatorKind;

public class Rule2 implements Rule {

    public Rule2() {
    }

    public boolean check(Object source) {
        return (source instanceof CombinedFragments) && (((CombinedFragments)
source).getOperator() == InteractionOperatorKind.ALT);
    }

    public PetriNet build(Object source, Transformer t) {
```

```
        PetriNet pn = new PetriNet(((CombinedFragments) source).getContext(),
((CombinedFragments) source).getID(), "ALT");
        Place p1 = new Place(new Mark(0));
        Place p2 = new Place(new Mark(0));
        for (int j = 0; j < ((CombinedFragments) source).getNumberOfFragments();
j++) {
            Transition trans = new Transition("ALT" + j);
            pn.addArc(new Arc(p1, trans, Arc.PLACE_TO_TRANSITION));
            PlaceHolder ph1 = new PlaceHolder(((CombinedFragments)
source).getContext(), ((CombinedFragments) source).getFragments()[j], "PH" + j);
            pn.addArc(new Arc(ph1, trans, Arc.TRANSITION_TO_PLACEHOLDER));
            Transition transEnd = new Transition("END-ALT" + j);
            pn.addArc(new Arc(ph1, transEnd, Arc.PLACEHOLDER_TO_TRANSITION));
            pn.addArc(new Arc(p2, transEnd, Arc.TRANSITION_TO_PLACE));
        }
        return pn;
    }

    public void setProperties(Object target, Object source, Transformer t) {
    }
}

package sitra;

import petrinet.Arc;
import petrinet.Mark;
import petrinet.PetriNet;
import petrinet.Place;
import petrinet.PlaceHolder;
import petrinet.Transition;
import sequencediagram.CombinedFragments;
import sequencediagram.InteractionOperatorKind;

public class Rule3 implements Rule {

    public Rule3() {
    }

    public boolean check(Object source) {
        return (source instanceof CombinedFragments) && (((CombinedFragments)
source).getOperator() == InteractionOperatorKind.OPT);
    }

    public PetriNet build(Object source, Transformer t) {
        PetriNet pn = new PetriNet(((CombinedFragments) source).getContext(),
((CombinedFragments) source).getID(), "OPT");
        Place p1 = new Place(new Mark(0));
        Place p2 = new Place(new Mark(0));
        for (int j = 0; j < ((CombinedFragments) source).getNumberOfFragments();
j++) {
            Transition trans = new Transition("OPT" + j);
            pn.addArc(new Arc(p1, trans, Arc.PLACE_TO_TRANSITION));
            PlaceHolder ph1 = new PlaceHolder(((CombinedFragments)
source).getContext(), ((CombinedFragments) source).getFragments()[j], "PH" + j);
            pn.addArc(new Arc(ph1, trans, Arc.TRANSITION_TO_PLACEHOLDER));
            Transition transEnd = new Transition("END-OPT" + j);
            pn.addArc(new Arc(ph1, transEnd, Arc.PLACEHOLDER_TO_TRANSITION));
            pn.addArc(new Arc(p2, transEnd, Arc.TRANSITION_TO_PLACE));
        }
        return pn;
    }

    public void setProperties(Object target, Object source, Transformer t) {
    }
}
```

```
package sitra;

import petrinet.Arc;
import petrinet.Mark;
import petrinet.PetriNet;
import petrinet.Place;
import petrinet.PlaceHolder;
import petrinet.Transition;
import sequencediagram.CombinedFragments;
import sequencediagram.InteractionOperatorKind;

public class Rule4 implements Rule {

    public Rule4() {
    }

    public boolean check(Object source) {
        return (source instanceof CombinedFragments) && (((CombinedFragments)
source).getOperator() == InteractionOperatorKind.BREAK);
    }

    public PetriNet build(Object source, Transformer t) {
        PetriNet pn = new PetriNet(((CombinedFragments) source).getContext(),
((CombinedFragments) source).getID(), "BREAK");
        Place p1 = new Place(new Mark(0));
        Place p2 = new Place(new Mark(0));
        Place pX = new Place(new Mark(0));
        Transition trans1 = new Transition("BREAK1");
        Transition trans2 = new Transition("BREAK2");
        Transition transEnd = new Transition("END-BREAK");
        pn.addArc(new Arc(p1, trans1, Arc.PLACE_TO_TRANSITION));
        pn.addArc(new Arc(p1, trans2, Arc.PLACE_TO_TRANSITION));
        PlaceHolder ph = new PlaceHolder(((CombinedFragments) source).getContext(),
((CombinedFragments) source).getFragments()[0], "PH");
        pn.addArc(new Arc(ph, trans1, Arc.TRANSITION_TO_PLACEHOLDER));
        pn.addArc(new Arc(ph, transEnd, Arc.PLACEHOLDER_TO_TRANSITION));
        pn.addArc(new Arc(pX, trans2, Arc.TRANSITION_TO_PLACE));
        pn.addArc(new Arc(p2, transEnd, Arc.TRANSITION_TO_PLACE));
        return pn;
    }

    public void setProperties(Object target, Object source, Transformer t) {
    }
}

package sitra;

import petrinet.Arc;
import petrinet.Mark;
import petrinet.PetriNet;
import petrinet.Place;
import petrinet.PlaceHolder;
import petrinet.Transition;
import sequencediagram.CombinedFragments;
import sequencediagram.InteractionOperatorKind;

public class Rule5 implements Rule {

    public Rule5() {
    }

    public boolean check(Object source) {
        return (source instanceof CombinedFragments) && (((CombinedFragments)
source).getOperator() == InteractionOperatorKind.PAR);
```

```
    }

    public PetriNet build(Object source, Transformer t) {
        PetriNet pn = new PetriNet(((CombinedFragments) source).getContext(),
((CombinedFragments) source).getID(), "PAR");
        Place p1 = new Place(new Mark(0));
        Place p2 = new Place(new Mark(0));
        Transition trans = new Transition("PAR");
        Transition transEnd = new Transition("END-PAR");
        pn.addArc(new Arc(p1, trans, Arc.PLACE_TO_TRANSITION));
        pn.addArc(new Arc(p2, transEnd, Arc.TRANSITION_TO_PLACE));
        for (int j = 0; j < ((CombinedFragments) source).getNumberOfFragments();
j++) {
            PlaceHolder ph1 = new PlaceHolder(((CombinedFragments)
source).getContext(), ((CombinedFragments) source).getFragments()[j], "PH" + j);
            pn.addArc(new Arc(ph1, trans, Arc.TRANSITION_TO_PLACEHOLDER));
            pn.addArc(new Arc(ph1, transEnd, Arc.PLACEHOLDER_TO_TRANSITION));
        }
        return pn;
    }

    public void setProperties(Object target, Object source, Transformer t) {
    }
}

package sitra;

import java.lang.reflect.Modifier;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Vector;

public class SimpleTransformerImpl implements Transformer {
       public SimpleTransformerImpl(List<Class<? extends Rule>> ruleTypes) {
              this.ruleTypes = ruleTypes;
       }
       Map<Class<? extends Rule>, Map<Object, Object>> mappings = new
HashMap<Class<? extends Rule>, Map<Object, Object>>();
       <S, T> Map<S, T> getRuleMappings(Class<? extends Rule<S, T>> rule) {
              Map<S, T> ruleMappings = (Map<S, T>) mappings.get(rule);
              if (ruleMappings == null) {
                     ruleMappings = new HashMap<S, T>();
                     mappings.put(rule, (Map<Object, Object>) ruleMappings);
              }
              return ruleMappings;
       }
       <S, T> void recordMaping(Class<? extends Rule<S, T>> rule, S source, T
target) {
              getRuleMappings(rule).put(source, target);
       }
       <S, T> T getExistingTargetFor(Class<? extends Rule<S, T>> rule, S source) {
              return getRuleMappings(rule).get(source);
       }
       <S, T> T applyRule(Rule<S, T> r, S source) {
              Class<? extends Rule<S, T>> ruleType = (Class<? extends Rule<S,
T>>)r.getClass();
              T target = getExistingTargetFor(ruleType, source);
              if (target == null) {
                     target = r.build(source, this);
                     recordMaping(ruleType, source, target);
                     r.setProperties(target,source,this);
              }
              return target;
       }
```

```
	// --- Transformer ---
	List<Class<? extends Rule>> ruleTypes;
	public List<Class<? extends Rule>> getRuleTypes() {
		if (this.ruleTypes == null) {
			this.ruleTypes = new Vector<Class<? extends Rule>>();
		}
		return this.ruleTypes;
	}
	public void addRuleType(Class<? extends Rule> ruleType) {
		getRuleTypes().add(ruleType);
	}
	List<Rule> getRules(Class<? extends Rule> ruleType) {
		List<Rule> rules = new Vector<Rule>();
		for (Class<? extends Rule> rt : getRuleTypes()) {
			if (ruleType.isAssignableFrom(rt)) {
				if (!Modifier.isAbstract(rt.getModifiers())) {
					try {
						rules.add(rt.newInstance());
					} catch (InstantiationException e) {
						e.printStackTrace();
					} catch (IllegalAccessException e) {
						e.printStackTrace();
					}
				}
			}
		}
		return rules;
	}
	public <S, T> T transform(Class<? extends Rule<S, T>> ruleType, S source) {
		try {
			List<Rule> rules = getRules(ruleType);
			//might be better to do this as an follows, but needs
assertions to be switched on
			assert !rules.isEmpty() : "No rule " + ruleType + " found in
transformer " + this;
			if (rules.isEmpty()) {
				System.err.println("No rule " + ruleType + " found in
transformer " + this);
			} else {
				for (Rule rule : rules) {
					Boolean b = false;
					try {
						b = rule.check(source);
					} catch (ClassCastException e) {}
					if (b) {
						return applyRule((Rule<S, T>) rule,
source);
					}
				}
			}
		} catch (Throwable t) {
			t.printStackTrace();
		}
		return null;
	}
	public <S, T> List<? extends T> transformAll(Class<? extends Rule<S, T>>
ruleType, List<? extends S> element) {
		List<T> targets = new Vector<T>();
		for (S s : element) {
			T o = transform(ruleType, s);
			targets.add(o);
		}
		return targets;
	}
	public Object transform(Object object) {
```

```
		return transform((Class)Rule.class, object);
	}
	public List<? extends Object> transformAll(List<? extends Object>
sourceObjects) {
		return transformAll((Class)Rule.class, sourceObjects);
	}
}

package sitra;

import java.util.List;

public interface Transformer {
	Object transform(Object object);
	List<? extends Object> transformAll(List<? extends Object> sourceObjects);
	<S, T> T transform(Class<? extends Rule<S, T>> ruleClass, S source);
	<S, T> List<? extends T> transformAll(Class<? extends Rule<S, T>> ruleClass,
List<? extends S> element);
}
```

REFERENCES

1. Sametinger, J., *Software Engineering with Reusable Components*. 1997: Springer.
2. Pan, J., *Software Testing*. Dependable Embedded Systems, 1999.
3. Pirzadeh, L., *Human Factors in Software Development: A Systematic Literature Review*. 2010, Chalmers University of Technology.
4. Spivey, J.M., *The Z Notation: a reference manual*. 2001: Prentice Hall (out of print, available at http://spivey.oriel.ox.ac.uk/~mike/zrm/).
5. AlloyAnalyzer, *Alloy Analyzer Website, http://alloy.mit.edu/beta/*2005.
6. Murata, T., *Petri Nets: Properties, Analysis and Applications.* Proceedings of the IEEE, 1989. **77**(4): p. 541-580.
7. OMG, *OMG Unified Modelling Language (UML) Superstructure 2.1, available at www.omg.org*. 2007.
8. Anastasakis, K., et al. *UML2Alloy: a Challenging Model Transformation*. in *ACM/IEEE 10th international conference on Model Driven Engineering Languages and Systems*. 2007.
9. Barkaoui, K., R. Ayed, and Z. Sbai, *Workflow Soundness Verification based on Structure Theory of Petri Nets.* International Journal of Computing & Information Sciences, 2007.
10. Christensen, S. and L. Petrucci, *Modular Analysis of Petri nets.* The Computer Journal, 2000. **43**(3): p. 224-242.
11. Christensen, S. and L. Petrucci. *Modular state space analysis of coloured Petri nets.* in *16th Int. Conf. Application and Theory of Petri Nets (ICATPN'95)*. 1995.
12. Esparza, J. and M. Silva, *On the analysis and synthesis of free choice systems.* Advances in Petri Nets, 1990.
13. Guerra, E. and J.d. Lara, *A Framework for the Verification of UML Models. Examples Using Petri Nets.* JISBD 2003, 2003: p. 325-334.
14. Javier, E., *Reachability in live and safe free-choice Petri nets is NP-complete.* Theor. Comput. Sci., 1998. **198**(1-2): p. 211-224.
15. Jiroveanu, G., R.K. Boel, and B. Bordbar, *Contextual Analysis of Partially Observable Large Petri Nets.* submitted to Journal of Discrete Event Dynamic Systems, 2005.
16. Valette, R., *Analysis of Petri Nets by Stepwise Refinement.* Journal of Computer and System Sciences, 1979.
17. Badouel, E. and P. Darondeau, *On the synthesis of General Petri Nets.* Inria Research Report 1996. **3025**.
18. Benatallah, B., et al. *A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling*. in *Business Process Management, International Conference, BPM 2003*. 2003: Springer Verlag.
19. Berthelot, G. *Transformation and decomposition of nets*. in *Advances in Petri nets*. 1986: Springer-Verlag.
20. Chao, D.Y. and D.T. Wang, *Knitting technique with TP-PT generations for Petri net synthesis*, in *IEEE International Conference on Intelligent Systems for the 21st Century*. 1995.
21. Chao, D.Y., M. Zhott, and D.T. Wang, *Extending Knitting Technique To Petri Net Synthesis Of Automated Manufacturing Systems*, in *Proceedings of the Third International Conference on Computer Integrated Manufacturing* 1992.

22. Esparza, J. and M. Silva, *Top-Down synthesis of live and bounded free choice nets.* Advances in Petri Nets, 1991.
23. Jeng, M.D. and F. DiCesare, *A review of synthesis techniques for Petri nets with applications to automated manufacturing systems.* IEEE Transactions on Systems, Man and Cybernetics, 1993.
24. Pancerz, K., *Synthesis of Petri Net Models: A Rough Set Approach.* Fundamenta Informaticae, 2003.
25. Wang, D.T. and D.Y. Chao, *Enhanced knitting technique to Petri net synthesis*, in *IEEE International Conference on Humans, Information and Technology*. 1994.
26. Zhou, M., D. F., and D.A. A., *A hybrid methodology for synthesis of Petri net models for manufacturing systems*, in *IEEE Transactions on Robotics and Automation*. 1992.
27. CPNTools, *Computer Tool for Coloured Petri Nets*, *http://wiki.daimi.au.dk/cpntools/*.
28. Bonet, P., et al., *PIPE v2.5: a Petri Net Tool for Performance Modeling*, in *XXXIii Conferencia Latinoaméricana de Informática*. 2007.
29. *AlPiNA : an Algebraic Petri Net Analyzer*. Available from: http://alpina.unige.ch/.
30. *GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets - GreatSPN*. Available from: http://www.di.unito.it/~greatspn/index.html.
31. *JFern, Java-based Petri Net framework*. Available from: http://sourceforge.net/projects/jfern/.
32. *LoLA - A Low Level Petri Net Analyser*. Available from: http://wwwteo.informatik.uni-rostock.de/ls_tpp/lola/.
33. *Netlab Petri Net tool*. Available from: http://www.irt.rwth-aachen.de/en/downloads/petri-net-tool-netlab.html.
34. *Petri Net World: Online Service for the International Petri Net Community*. Available from: www.informatik.uni-hamburg.de/TGI/PetriNets.
35. *TINA - Time Petri Net Analyzer*. Available from: http://homepages.laas.fr/bernard/tina/.
36. Desel, J. and J. Esparza, *Free Choice Petri Nets*. 1995: Cambridge University Press.
37. Esparza, J. and M. Silva, *A Polynomial-Time Algorithm to Decide Liveness of Bounded Free Choice Nets.* Theoretical Computer Science, 1992. **102**: p. 185-205.
38. Baccelli, F., S. Foss, and B. Gaujal, *Free Choice Petri Net: an Algebraic Approach.* IEEE Trans. on Automatic Control, 1996.
39. Baccelli, F.o., B. Gaujal, and S. Foss, *Structural, temporal and stochastic properties of unbounded free-choice Petri nets*. 2006, HAL - CCSD.
40. Küster-Filipe, J., *Modelling concurrent interactions.* Theoretical Computer Science, 2006. **351**(2): p. 203-220.
41. McMillan, K.L., *A technique of state space search based on unfolding.* Form. Methods Syst. Des., 1995. **6**(1): p. 45-65.
42. Winskel, G., *An introduction to event structures*, in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*. 1989, Springer-Verlag.
43. Liang, H., et al. *A general approach for scenario integration*. in *11th international conference on Model Driven Engineering Languages and Systems*. 2008: Springer-Verlag.
44. Liang, H., *Sequence Diagram Integration via Typed Graph: Theory and Implementation*, in *School of Computing*. 2009, Queen's University: Kingston, Ontario, Canada.

45. Bowles, J.K.F. and B. Bordbar. *A Formal Model for Integrating Multiple Views*. in *Application of Concurrency to System Design (ACSD)*. 2007.
46. Klein, J., L. Hélouët, and J.-M. Jézéquel, *Semantic-based weaving of scenarios*, in *AOSD-Europe : European Network of Excellent on Aspect-oriented Software Development*. 2006.
47. Krüger, I.H., *Distributed System Design with Message Sequence Charts*. 2000, Technische Universität München.
48. Jeng, M.D. and F. DiCesare, *A review of synthesis techniques for Petri nets with applications to automated manufacturing systems.* IEEE Transactions on Systems, Man and Cybernetics, 1993.
49. Xu, J., M. Woodside, and D. Petriu, *Performance Analysis of a Software Design Using the UML Profile for Schedulability, Performance, and Time*, in *Computer Performance*. 2003, Springer Berlin / Heidelberg.
50. Wang, J., *Timed Petri Nets: Theory and Application*. 1998: Springer.
51. Sannella, D., *A Survey of Formal Software Development Methods*, in *Tech. Rept. ECS-LFCS-88-56*. 1988, Edinburgh University.
52. Medvidovic, N., R.F. Gamble, and D.S. Rosenblum, *Towards Software Multioperability: Bridging Heterogeneous Software Interoperability Platforms*, in *Fourth International Software Architecture Workshop (ISAW-4)*. 2000: Limerick, Ireland.
53. Jackson, D., *Software Abstractions Logic, Language, and Analysis*. 2006: MIT press.
54. Georg, G., et al., *An Aspect-Oriented Methodology for Developing Secure Applications.* Journal of Information and Software Technology, 2009. **51**(5): p. 846-864.
55. Reddy, Y.R., et al., *Directives for Composing Aspect-Oriented Design Class Models* in *Transactions on Aspect-Oriented Software Development I* 2006, Springer Berlin / Heidelberg.
56. Klein, J., F. Fleurey, and J.-M. Jézéquel, *Weaving Multiple Aspects in Sequence Diagrams*, in *Transactions on Aspect-Oriented Software Development III*, S.B. Heidelberg, Editor. 2007.
57. Klein, J. and J. Kienzle, *Reusable Aspect Models*, in *11th Workshop on Aspect-Oriented Modeling*. 2007: Nashville, TN, USA.
58. Yakovlev, A.V., et al., *Modelling, Analysis and Synthesis of Asynchronous Control Circuits Using Petri Nets.* INTEGRATION: the VLSI Journal 1996. **21**: p. 143-170.
59. Agerwala, T. and Y.-C. Choed-Amphai, *A synthesis rule for concurrent systems*, in *ACM IEEE Design Automation Conference*. 1978.
60. Ameedeen, M.A., B. Bordbar, and R. Anane, *Model Interoperability via Model Driven Development* accepted for publication in Journal of Computer and System Sciences, 2010.
61. Ameedeen, M.A., B. Bordbar, and R. Anane, *A Model Driven Approach to Analysis of Timeliness Properties*, in *Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA 2009) (to appear)*. 2009.
62. Ameedeen, M.A. and B. Bordbar, *A Model Driven Approach to Represent Sequence Diagrams as Free Choice Petri Nets*, in *12th International IEEE Enterprise Distributed Object Computing Conference (EDOC)*. 2008: München, Germany. p. 213 - 221.
63. Pender, T., *UML Bible*. 2003, Indianapolis, {IN}: Wiley Publishing.
64. Miles, R. and K. Hamilton, *Learning UML 2.0*. 2006: O'Reilly Media, Inc.

65. Harel, D., *Statecharts: a visual formalism for complex systems.* Science of Computer Programming, 1987. **8**(3): p. 231-274.
66. Mauw, S. and M. Reniers, *An algebraic semantics of basic message sequence charts.* The Computer Journal, 1994. **37**: p. 269-277.
67. Alur, R., E. Kousha, and Y. Mihalis, *Inference of message sequence charts*, in *Proceedings of the 22nd international conference on Software engineering*. 2000, ACM: Limerick, Ireland.
68. Baker, P., et al., *Model-Driven Testing: Using the UML Testing Profile*. 2007: Springer-Verlag New York, Inc.
69. Dufourd, C., A. Finkel, and P. Schnoebelen, *Reset Nets Between Decidability and Undecidability*, in *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*. 1998, Springer-Verlag.
70. Marsan, M.A., *Stochastic Petri nets: an elementary introduction*, in *Advances in Petri Nets 1989, covers the 9th European Workshop on Applications and Theory in Petri Nets-selected papers*. 1990, Springer-Verlag.
71. Jensen, K., *Coloured Petri Nets (2nd ed.): basic concepts, analysis methods and practical use: volume 1*. 1996: Springer-Verlag. 234.
72. Guan, S.-U. and S.-S. Lim, *Modeling with enhanced prioritized Petri nets: EP-nets.* Computer Communications, 2002. **25**(8): p. 812-824.
73. Dawis, E.P., J.F. Dawis, and W.-P. Koo, *Architecture of Computer-based Systems using Dualistic Petri Nets*, in *IEEE International Conference on Systems, Man, and Cybernetics*. 2001.
74. Jensen, K., L.M. Kristensen, and L. Wells, *Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems.* International Journal on Software Tools for Technology Transfer (STTT), 2007.
75. Vanhatalo, J., H. Volzer, and F. Leymann, *Faster and More Focussed Control-Flow Analysis for Business Process Models Through SESE Decomposition*, in *Fifth International Conference on Service Oriented Computing*. 2007, Springer: Vienna, Austria. p. 43-55.
76. Kovalyov, A. and J. Esparza, *A Polynomial Algortihm to Compute the Concurrency Relation of Free-Choice Signal Transition Graphs*, in *International Workshop on Discrete Event Systems*. 1995.
77. *Business Process Management (BPM), Workflow and DMS Solutions by COSA.* Available from: http://www.cosa.nl/Homeen.html.
78. *HELENA - a High LEvel Net Analyzer*. Available from: http://helena.cnam.fr/.
79. Anschuetz, H. *HPSim*. Available from: http://www.winpesim.de/3.html.
80. *INA - Integrated Net Analyzer*. Available from: http://www2.informatik.hu-berlin.de/~starke/ina.html.
81. *Maria: The Modular Reachability Analyzer*. Available from: http://www.tcs.hut.fi/Software/maria/.
82. *PetriSim - Discrete Simulation Environment*. Available from: http://staff.um.edu.mt/jskl1/petrisim/.
83. *TimeNET (timed net evaluation tool).* Available from: http://www.tu-ilmenau.de/fakia/8086.html.
84. Stahl, T. and M. Volter, *Model Driven Software Development; technology engineering management*. 2006: Wiley.
85. MDA, *Model Driven Architecture, Object Management Group www.omg.org/mda/.* 2005.

86. MOF. *Meta Object Facility (MOF) 2.0 Core Specification, Object Management Group, available at www.omg.org,*. 2004; Available from: http://www.omg.org.
87. ATLAS, *ATLAS, Université de Nantes, http://www.sciences.univ-nantes.fr/lina/atl/*. 2005.
88. kermeta, *Triskell Metamodelling Kernel, www.kermeta.org*. 2005.
89. Akehurst, D.H., et al. *SiTra: Simple Transformations in Java*. in *ACM/IEEE 9TH International Conference on Model Driven Engineering Languages and Systems*. 2006.
90. Blaha, M. and J. Rumbaugh, *Object-Oriented Modeling and Design with UML, 2/E* 2005: Prentice Hall.
91. Gigch, J.P.V., *System design modeling and metamodeling*. 1991: Springer.
92. Loucopoulos, P. and V. Karakostas, *System Requirements Engineering*. 1995: McGraw Hill.
93. Smullyan, R.M., *First-Order Logic*. 1995, New York: Dover Publications.
94. Wieland, P., F. Høgberg, and K. Strømseng, *Enhancements in Software Project Risk Management*, in *Reliable Software Technologies Ada-Europe 2000*. 2000, Springer Berlin / Heidelberg.
95. van der Aalst, W.M.P., *The Application of Petri Nets for Workflow Management.* The Journal of Circuits, Systems and Computers, 1998. **8**(1): p. 21-66.
96. Villa, F. and R. Costanza, *Design of Multi-Paradigm Integrating Modelling Tools for Ecological Research.* Environmentla Modelling & Software, 2000. **15**.
97. *The OsMoSys approach to multi-formalism modeling of systems.* Software and Systems Modeling (SoSyM), 2004. **3**: p. 68-81.
98. Vangheluwe, H. and E. Kerckhoffs, *Computer automated modelling of complex systems*, in *15th European Simulation Multi-conference (ESM)*. 2001: Prague, Czech Republic.
99. de Lara, J. and H. Vangheluwe, *Computer Aided Multi-paradigm Modelling to Process Petri-Nets and Statecharts*, in *First International Conference on Graph Transformation*. 2002.
100. de Lara, J., H. Vangheluwe, and M. Alfonseca, *Computer Aided Multi-Paradigm Modelling of Hybrid Systems with AToM3*, in *Summer Computer Simulation Conference: Society for Computer Simulation International (SCS)*. 2003: Montreal, Canada.
101. Mosterman, P.J. and H. Vangheluwe, *Computer automated multi paradigm modeling in control system design*, in *IEEE International Symposium on Computer-Aided Control System Design*. 2002, IEEE Computer Society Pres: Anchorage, Alaska.
102. Mosterman, P.J. and H. Vangheluwe, *Guest editorial: Special issue on computer automated multi-paradigm modeling.* ACM Transactions on Modeling and Computer Simulation (TOMACS), 2002. **12**(4): p. 249-255.
103. Vangheluwe, H., J.d. Lara, and P.J. Mosterman, *An introduction to multi-paradigm modelling and simulation*, in *AI, Simulation and Planning in High Autonomy Systems (AIS'2002)*. 2002: Lisboa, Portugal.
104. Czarnecki, K. and S. Helsen, *Feature-based survey of model transformation approaches.* IBM Systems Journal, 2006. **45**.
105. Varró, D. and A. Pataricza, *Generic and Meta-Transformations for Model Transformation Engineering*, in *7th International Conference on the Unified Modeling Language*. 2004: Lisbon, Portugal

106. Varró, D., G. Varró, and A. Pataricza, *Designing the Automatic Transformation of Visual Languages.* Science of Computer Programming 2002. **44**.
107. Akehurst, D.H., W.G. Howells, and K.D. McDonald-Maier. *Kent Model Transformation Language*. in *Model Transformations in Practice Workshop, part of MoDELS 2005*. 2005. Montego Bay,Jamaica.
108. Jouault, F. and I. Kurtev. *Transforming Models with ATL* in *Model Transformations in Practice Workshop at MoDELS*. 2005. Montego Bay, Jamaica.
109. kermeta. *Triskell Metamodelling Kernel*. 2005; Available from: www.kermeta.org.
110. Akehurst, D.H., et al. *SiTra: Simple Transformations in Java*. in *ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences)*. 2006. Genova, Italy.
111. SiTra, *Simple Transformer (SiTra): an MDE tool, www.cs.bham.ac.uk/~bxb/SiTra.html*. 2006.
112. OMG, *MOF 2.0 Query/View/Transformation (QVT) Specification, available at www.omg.org*. 2008.
113. Henkler, S. and M. Hirsch, *A Multi-Paradigm Modeling Approach for Reconfigurable Mechatronic Systems*, in *International Workshop on Multi-Paradigm Modeling: Concepts and Tools (MPM06), Satellite Event of the the 9th International Conference on Model-Driven Engineering Languages and Systems MoDELS/UML2006*. 2006: Genova, Italy.
114. Burmester, S., H. Giese, and M. Tichy, *Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML* in *Lecture Notes in Computer Science* 2005, Springer Berlin / Heidelberg.
115. UML2Alloy. *A tool for analysis of UML model via Alloy, available at www.cs.bham.ac.uk/~bxb/old_UML2Alloy.html*. 2005.
116. OMG. *UML 2.0 OCL Specification*. Document Id: ptc/03-10-14 2003; OMG Final Adopted Specification:[Available from: http://www.omg.org/docs/ptc/05-06-06.pdf.
117. Kim, S.-K., *A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques*. 2002, University of Queensland: Brisbane, Australia.
118. Marcano, R. and N. Lévy, *Transformation Rules of OCL Constraints into B Formal Expressions*, in *5th International Conference on the Unified Modeling Language*. 2002: Dresden, Germany.
119. Snook, C. and M. Butler, *UML-B: Formal modelling and design aided by UML*, in *ACM Transactions on Software Engineering and Methodology*. 2006.
120. Abrial, J.-R., *The B-book: Assigning Programs to Meanings*. 1996: Cambridge University Press.
121. Evans, A.F., Robert & Grant, Emanuel. *Towards Formal Reasoning with UML Models*. in *Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics*. 1999.
122. Kim, D., et al. *A UML-Based Metamodeling Language to Specify Design Patterns*. 2003; Available from: http://www.cs.colostate.edu/~georg/aspectsPub/WISME03-dkk.pdf.
123. Muscholl, A. and D. Peled, *Analyzing message sequence charts*, in *2nd Workshop on SDL and MSC*. 2000: Grenoble, France.
124. Kühn, H., M. Murzek, and F. Bayer, *Horizontal Business Process Model Interoperability using Model Transformation*, in *INTEREST'2004 Workshop at ECOOP 2004*. 2004: Oslo, Norway.

125. Bertolini, D., et al., *A Tropos Model-Driven Development Environment*, in *18th Conference on Advanced Information Systems Engineering (CAiSE-06)*. 2006, Springer Verlag: Luxembourg.
126. Méry, D., et al., *Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques*, in *Integrated Formal Methods*, Springer Berlin / Heidelberg. p. 183-198.
127. Sousa, V.N.d.S.d., *Model driven development implementation of a control systems user interfaces specification tool*. 2009, Universidade Nova de Lisboa.
128. Poernomo, I., *Proofs-as-Model-Transformations*, in *Proceedings of the 1st international conference on Theory and Practice of Model Transformations*. 2008, Springer-Verlag: Zurich, Switzerland.
129. Poernomo, I. and J. Terrell, *Correct-by-construction model transformations from partially ordered specifications in Coq*, in *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, Springer-Verlag: Shanghai, China.
130. Fiorentini, C., et al., *A constructive approach to testing model transformations*, in *Proceedings of the Third international conference on Theory and practice of model transformations*, Springer-Verlag: M\&\#225;laga, Spain.
131. Lúcio, L., B. Barroca, and V. Amaral, *A technique for automatic validation of model transformations*, in *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, Springer-Verlag: Oslo, Norway.
132. Kuster, J.M., K. Ryndina, and R. Hauser, *A Systematic Approach to Designing Model Transformations*, IBM Research GmbH: Zurich Research Laboratory.
133. UML, *UML Superstructure 2.0, Object Management Group, available at www.omg.org*. 2003.
134. Booch, G., J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. 1998: Addison Wesley.
135. Radu, G. and A.S. Scott, *Safety-Liveness Semantics for UML 2.0 Sequence Diagrams*, in *Proceedings of the Fifth International Conference on Application of Concurrency to System Design*. 2005, IEEE Computer Society.
136. Cavarra, A. and J. Küster-Filipe, *Combining Sequence Diagrams and OCL for Liveness*. Electronic Notes in Theoretical Computer Science, 2005. **115**: p. 19-38.
137. OMG. *XML Metadata Interchange (XMI), v2.0*. 2005; Available from: http://www.omg.org.
138. W3C, *Extensible Markup Language (XML) 1.0, Third Edition, W3C Recommendation*. 2004.
139. ArgoUML, *ArgoUML web site, sourceforge.net/projects/argouml*. 2005.
140. Poseidon. *Poseidon for UML, from Gentleware, www.gentleware.com/*. 2006.
141. SDMetrics. *SDMetrics: The Software Design Metrics tool for the UML*. Available from: http://www.sdmetrics.com.
142. Schiller, J.H., *Mobile Communications*. 2003: Pearson Education.
143. Pfleeger, S.H. and J.M. Atlee, *Software Engineering - Theory and Practice*. 3rd Edition ed. 2006: Prentice Hall.
144. Chao, D.Y. and D.T. Wang, *Petri net synthesis and synchronization using knitting technique.* Systems, Man, and Cybernetics, 1994.
145. Esparza, J., *Reduction and synthesis of live and bounded free choice Petri nets.* Information and Computation, 1994.

146. Haddad, S. *A reduction theory for coloured Peti nets*. in *Advances in Petri nets*. 1989: Springer-Verlag.
147. SUZUKI, I. and T. MURATA, *A method for stepwise refinement and abstraction of Petri nets.* Journal of computer and system sciences, 1983.
148. Graf, S., I. Ober, and I. Ober, *Timed annotations in UML.* International Journal on Software Tools for Technology Transfer (STTT), 2005.
149. Douglass, B.P., *Real-Time UML Second Edition. Developing Efficient Objectsfor Embedded Systems*. Object Technology Series. 1999: Addison Wesley.
150. Bordbar, B. and R. Anane. *An Architecture for Automated QoS Resolution in Wireless Systems*. in *Proceeding of the IEEE International Workshop on Web and Mobile Information Systems (WAMIS)*. 2005.
151. OMG, *Response to the OMG RFP for Schedulability, Performance and Time.* 2001.
152. Knapp, A., S. Merz, and C. Rauh, *Model checking-timed UML state machines and collaborations*, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*. 2002, Springer: Oldenburg, Germany.
153. Roubtsova, E.E., et al., *Real-Time Systems: Specification of Properties in UML*, in *ASCI 2001 Conference*. 2001: Heijen, The Netherlands.
154. Li, X. and J. Lilius, *Timing Analysis of UML Sequence Diagrams*. 1999, Turku Centre for Computer Science.
155. Firley, T., et al., *Timed Sequence Diagrams and tool-based analysis : A case study*, in *UML '99 : the unified modeling language : beyond the standard* 1999, Springer, Berlin: Fort Collins.
156. Lund, M.S., *Operational analysis of sequence diagram specifications*. 2007, The University of Oslo: Oslo.
157. Haugen, Ø., et al., *Why Timed Sequence Diagrams Require Three-Event Semantics*, in *Scenarios: Models, Transformations and Tools*, S.B. Heidelberg, Editor. 2005.
158. Kabous, L. and W. Nebel, *Modeling Hard Real Time Systems with UML The OOHARTS Approach*, in *«UML»'99 — The Unified Modeling Language*. 1999, Springer Berlin / Heidelberg.
159. Flake, S. and W. Mueller, *A UML Profile for Real-Time Constraints with the OCL*, in *Proceedings of the 5th International Conference on The Unified Modeling Language*. 2002, Springer-Verlag.
160. Lanusse, A., S. Gérard, and F. Terrier, *Real-Time Modeling with UML: The ACCORD Approach*, in *The Unified Modeling Language. «UML»'98: Beyond the Notation*. 1999, Springer Berlin / Heidelberg.
161. Seemann, J. and J.W.v. Gudenberg, *Extension of UML Sequence Diagrams for Real-Time Systems*, in *The Unified Modeling Language. «UML»'98: Beyond the Notation*. 1999, Springer Berlin / Heidelberg.
162. Störrle, H., *Trace Semantics of UML 2.0 Interactions*. 2004, University of Munich.
163. Douglass, B.P., *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks and Patterns*. Object Technology Series. 1999: Addison Wesley.
164. Petriu, D.C. and H. Shen, *Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications*, in *Computer Performance Evaluation: Modelling Techniques and Tools*. 2002, Springer Berlin / Heidelberg.
165. Petriu, D.C., H. Shen, and A. Sabetta, *Performance Analysis of Aspect-Oriented UML Models.* Software and Systems Modeling, 2007. **Vol. 6**: p. 453-471.

166. Skene, J. and W. Emmerich, *Model Driven Performance Analysis of Enterprise Information Systems.* Electronic Notes in Theoretical Computer Science, 2003. **82**(6): p. 147-157.
167. Lindemann, C., et al., *Performance analysis of time-enhanced UML diagrams based on stochastic processes*, in *Proceedings of the 3rd international workshop on Software and performance*. 2002, ACM: Rome, Italy.
168. Simonetta, B., et al., *Model-Based Performance Prediction in Software Development: A Survey*. IEEE Transactions on Software Engineering, 2004. **30**: p. 295-310.
169. Viehl, A., et al., *Formal performance analysis and simulation of UML/SysML models for ESL design*, in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*. 2006, European Design and Automation Association: Munich, Germany.
170. OMG, *UML Profile for Schedulability, Performance and Time Specification.* 2002.
171. Budinsky, F., et al., *Eclipse Modeling Framework: A Developer's Guide*. 2003: Addison Wesley.