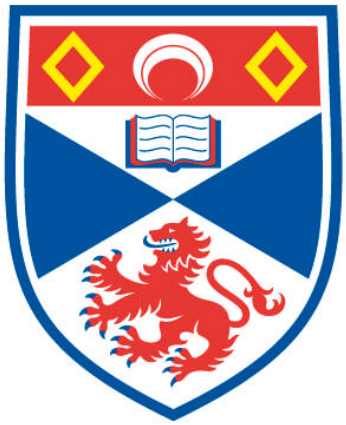
CORRECT MODEL-TO-MODEL TRANSFORMATION FOR FORMAL VERIFICATION

Dulani A. Meedeniya

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2013

Correct Model-to-Model Transformation for Formal Verification

Dulani A. Meedeniya

PhD Thesis

A Thesis submitted

for the degree of Doctor of Philosophy

School of Computer Science

University of St Andrews

May 2013

Abstract

Modern software systems have increasingly higher expectations on their reliability, in particular if the systems are critical and real-time. The development of these complex software systems requires strong modelling and analysis methods including quantitative modelling and formal verification.

Unified Modelling Language (UML) is a widely used and intuitive graphical modelling language to design complex systems, while formal models provide a theoretical support to verify system design models. However, UML models are not sufficient to guarantee correct system designs and formal models, on the other hand, are often restrictive and complex to use. It is believed that a combined approach comprising the advantages of both models can offer better designs for modern complex software development needs.

This thesis focuses on the design and development of a rigorous framework based on Model Driven Development (MDD) that facilitates transformations of non-formal models into formal models for design verification. This thesis defines and describes the transformation from UML2 sequence diagrams to coloured Petri nets and proves syntactic and semantic correctness of the transformation. Additionally, we explore ways of adding information (time, probability, and hierarchy) to a design and how it can be added onto extensions of a target model. Correctness results are extended in this context.

The approach in this thesis is novel and significant both in how to establish semantic and syntactic correctness of transformations, and how to explore semantic variability in the target model for formal analysis. Hence, the motivation of this thesis establishes: the UML behavioural models can be validated by correct transformation of them into formal models that can be formally analysed and verified.

Acknowledgements

I owe my gratitude to many people who have made this dissertation possible.

My deepest gratitude is to my first supervisor, Dr. Juliana Bowles for the opportunities and the support that she has given me. I have been fortunate to have her as a supervisor. Juliana always guided me to the right direction and to continue my research in a systematic way. Specially, her patience and valuable effort in driving me to do an affluent research is highly appreciated.

I would like to thank my second supervisor, Dr. Dharini Balasubramaniam, and my reviewer Dr. Graham Kirby, for their friendly support and feedback. They have been there to help, listen and give encouraging advices.

I would like to acknowledge my friends and colleagues through the years for supporting me many ways. It was a wonderful experience to be with them.

Also, I appreciate all the support from The Scottish Informatics and Computer Science Alliance (SICSA) for funding my PhD studies.

I am grateful to my family for their love and tremendous support. They encouraged and helped me in whatever way they could.

Finally, I appreciate everyone who helped me for making me who I am today.

Thank you.

Candidate's declarations

I Dulani A. Meedeniya, hereby certify that this thesis, which is approximately 77200 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree. I was admitted as a research student and a candidate for the degree of Doctor of Philosophy in January 2009; the higher study for which this is a record was carried out in the University of St Andrews between 2009 and 2013.

Date Signature of candidate

Supervisor's declarations

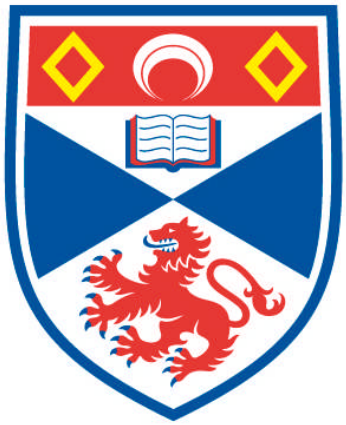
I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in Computer Science in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date Signature of supervisor

CORRECT MODEL-TO-MODEL TRANSFORMATION FOR FORMAL VERIFICATION

Dulani A. Meedeniya

A Thesis Submitted for the Degree of MPhil
at the
University of St Andrews



2013

Permission for electronic publication

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona file library or research worker, that my thesis will be electronically accessible for personal or research use unless exempt by award of an embargo as requested below, and that the library has the right to migrate my thesis into new electronic forms as required to ensure continued access to the thesis. I have obtained any third-party copyright permissions that may be required in order to allow such access and migration, or have requested the appropriate embargo below.

The following is an agreed request by candidate and supervisor regarding the electronic publication of thesis:

"Access to printed copy and electronic publication of thesis through the University of St Andrews."

Date

Signature of candidate

Signature of supervisor

Contents

List of Figures

List of Tables

Abbreviations

MDD	Model Driven Development
OMG	Object Management Group
UML	Unified Modelling Language
SD	Sequence Diagram
IOD	Interaction Overview Diagram
OCL	Object Constraint Language
CPN	Coloured Petri net
TCPN	Timed Coloured Petri net
SCPN	Stochastic Coloured Petri net
HCPN	Hierarchical Coloured Petri net
MOF	Meta Object Facility
XML	Extensible Mark up Language
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query-View-Transform
TGG	Triple Graph Grammar
SiTra	Simple Transformer

1 Chapter 1: Introduction

Modern software systems have increasingly higher expectations on their performance, security, availability and/or reliability, specially if the systems are critical. Therefore, there is a natural need for techniques that can help with developing systems to be dependable. A software system is dependable if the services provided by the system can be trusted [Gorton, 2006, Sommerville, 2007, Cohen et al., 1986, Avizienis et al., 2001, Avizienis et al., 2004, Bondavalli et al., 2005, Abdallah et al., 2005, Meyer, 2006]. The development of these software systems requires strong modelling and analysis methods including formal modelling and verification [Garousi, 2010, Naumenko and Wegmann, 2002, Rafe et al., 2009, Emadi and Shams, 2009b, Haugen et al., 2005, Shen et al., 2008a, Limaa et al., 2009, Cabot et al., 2008, Kounev and Buchmann, 2006, Lakos and Petrucci, 2004, Merseguer and Campos, 2004, Mallet et al., 2006, Tang et al., 2010].

Consequently, model-based software development, also known as Model Driven Development (MDD), is becoming a mainstream practice in software development. The MDD approach focuses on creating models and exploring their abstract representations towards concrete implementations. Models help to cope with the large scale and complexity of software systems by specifying the structural and behavioural aspects of the system and providing a means of communication between domain experts, analysts, designers and developers.

The use of the object-oriented Unified Modelling Language (UML), although popular in industry and an intuitive mechanism to design complex systems, is not sufficient to guarantee correct design. If a design can be formalised, we can take advantage of the theoretical support available in the underlying formalism to check system models and guarantee their properties such as correctness, completeness and performance. However, such practices

are not often widely used beyond academic research due to the inherent complexity of formal methods. Moreover, formal models and techniques are often restrictive to use by non-expert users.

Consequently, there is an increasing need to combine the benefits of popular design approaches and formal models to contribute to better software products. In addition, combining MDD with formal methods also requires model transformations to be proved correct and complete. This is important to ensure that the results of formal analysis will not be invalidated by erroneous transformations as developers cannot distinguish whether an error is in the design or in the transformation.

This thesis designs and develops a novel formal model transformation framework that brings formal methods more naturally into MDD. Our model transformation framework consists of a family of transformations from non-formal UML behavioural models to different formal models, which can be analysed in various ways to validate the original design models. The framework solution is convenient and appropriate for efficient system design and analysis, as well as adaptable and scalable for future system needs.

This chapter is organized into a number of sections. Collectively, these describe the problem space that the work addresses, the thesis contribution and a guide to the content of each chapter.

1.1 Motivation and Research Overview

Modern software systems in most domains are complex, large-scale, and often critical. It is hard to develop such systems when taking into account their real-time and stochastic requirements. The development of these complex software systems requires strong modelling and analysis methods.

One possible solution to build systems with complex requirements may be

primarily to concentrate efforts on modelling the system design, analyse and then develop the software system from these models. This implies that we need formal techniques to guarantee that a system model is complete and correct, and thus develop a consistent, software system with respect to its specification. Software development using a MDD approach addresses some of these issues by providing abstract mechanisms and the infrastructure necessary to develop these software systems.

Formal modelling is a technique that supports the validation and verification of software models in the design stage. Although, UML is a widely used technique to design structural and behavioural aspects of a software system, the direct applicability of formal techniques is not possible. The transformation of non-formal models such as UML to formal models enables possible formal analysis of the system model, at the design stage. This validates the original UML models and leads to complete and correct system development. Different aspects of the system model can be represented using different formalisms by defining a precise set of transformation rules. In order to enable formal analysis, model-to-model (M2M) transformations can be defined where the target model is the underlying mathematical model used for a particular analysis approach.

The work done in this thesis borrows ideas from MDD to construct an environment where system models can be analysed and validated. This thesis establishes a formal model transformation and integration framework for software system design (Figure 1.1). With this framework, it is possible to apply model transformations and extensions to other formal models, and reuse the transformations which provide a productive working scenario by saving project cost and time.

This thesis uses coloured Petri nets (CPNs) as the synthesised formal model

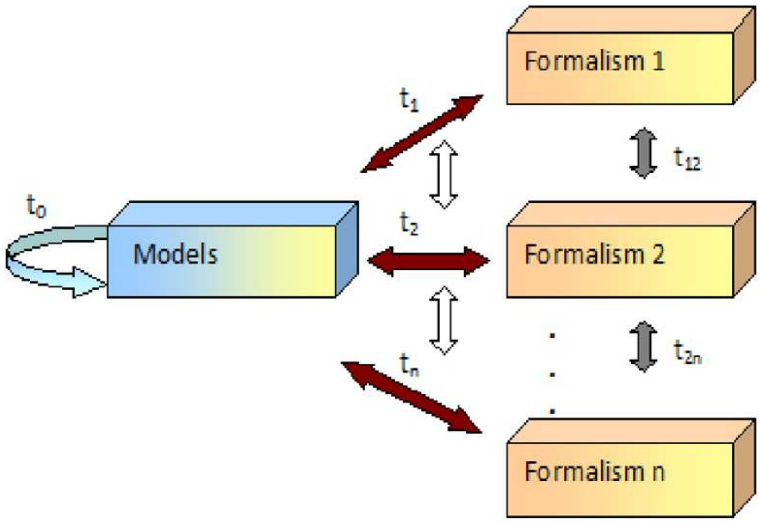


Figure 1.1: The model transformation framework.

with rich variants, which can produce a useful range of verification options. CPNs are a well-known formal model with a rich theory and practice, which are well suited for our approach when transforming object-oriented models. Our approach applies model transformations from UML sequence diagrams (SDs) to variants of CPNs, to enable different possible analyses of the model. It assumes that behavioural aspects of systems are modelled at the design level using SDs and our approach obtains a formal representation by transforming SDs into variants of CPNs including timed coloured Petri nets (TCPN), stochastic coloured Petri nets (SCPN) and hierarchical coloured Petri-nets (HCPN).

The flexibility of this model transformation framework lies in the incremental nature of the transformations. In particular, given a SD and its corresponding CPN, in case when the SD extends with time and stochastic annotations the corresponding CPN variants can be generated by incrementally applying the specific variant rules on the original CPN. Apart from the strongly consistent transformation of UML models into variants of CPN, this thesis considers

model composition, as part of a general model transformation framework. For instance, this thesis defines the model composition between SDs with reference behaviour, SDs and Interaction Overview Diagrams (IODs), and CPNs of hierarchical nature.

Further, this research proves the syntactical and semantical correctness of the defined model transformation, elaborates the applicability of the transformation using example case-studies, and develops a prototype tool, as part of this research contribution.

1.2 Thesis Objectives and Methodology

The thesis statement of this dissertation is that:
"UML behavioural models can be validated by correct transformation into formal models that can be formally analysed and verified".

Given the motivation of developing a framework for M2M transformations, this research has the following objectives to:

Ob1: Define strongly consistent languages for UML SD and CPN

Ob2: Define formal transformation rules for the mapping of UML2 SDs into variants of CPNs

Ob3: Define model composition rules between models with reference behaviour

Ob4: Support flexible model transformation framework with an incremental nature of the transformations that enables the analysis of sub-interactions

Ob5: Prove the correctness and completeness of defined M2M transformations

Ob6: Explore the applicability and the implementation ability of the defined M2M transformations

In order to achieve the objectives, this thesis utilises a number of methodologies. First a survey of software design models and related formal model transformation approaches was conducted in order to identify possible models for M2M transformations. UML 2 sequence diagram ([Arlow and Neustadt, 2005, Douglass, 2004, OMG, 2011a]) is identified as the non-formal design model, while CPN models ([Jensen, 1997a, Kristensen et al., 2004, Jensen and Kristensen, 2009]) with its variants are chosen as the underlying formal models for this thesis.

Secondly, formal representations were defined for each identified model in order to state the meaning of the model unambiguously and to enable formal verification. Consequently, strongly consistent languages were defined for SD and CPN that establishes a direct correspondence between the elements.

Thirdly, formal exogenous transformation ([T.Mens and Grop, 2006]) rules were defined for the transformations from a SD to a CPN capturing both general and complex behaviours. Partial, incremental and parametric transformations were defined between models considering different regions and variants. Further, these rules were extended to define model composition and integration rules.

Fourthly, the defined model transformations were evaluated for their correctness and completeness using declarative and operational approaches. In particular, mathematical proof techniques were used to prove the semantic correctness of the model transformation. Further, example-based case studies with manual analysis of the synthesised model have been considered to show the applicability of the defined transformations in practical use.

Finally, a prototype tool was implemented to explore the possibility of developing a model transformation framework. Meta-models of each model and the formal transformation rules were incorporated into the core implementa-

tions. The tool provides a user interface using NetBeans IDE and facilitates future extensions.

1.3 Thesis Contributions

The contribution of this thesis highlights the successful achievements of the objectives. We believe this thesis has significantly contributed to uplift M2M transformation framework, providing valuable outcomes for future research of MDD and in particular establishing the semantic correctness of model transformation.

Our contributions are of the following major forms:

- Scholarly publications resulting in the timely dissemination of the research findings

- Implementation of a prototype tool aiming for a proof of concept and prospects for practical tool development

- Development of the model transformation framework that supports and scaffolds related research contributing significantly to the MDD based software development.

The primary contribution of this thesis is the formal representations of UML2 SD, IOD, CPN and its variants taking into account the associated complex behaviours with time, stochastic and hierarchical variations. As the major contribution, formal rules were defined for model transformations, compositions and integrations while proving the syntactic and semantic correctness of the transformations. In particular, the flexibility of the model transformation framework lies in the incremental nature of the transformations. A part of this framework is implemented in a prototype tool and evaluated using example case studies.

Some of the research presented in this thesis has been published in the following journal and conference papers:

P1: Bowles, J. and Meedeniya, D. (2012). Parametric transformations for flexible analysis. In Proceedings of the 19th Asia Pacific Software Engineering Conference (APSEC '12), pages 634-643. IEEE Computer Society.

P2: Bowles, J. K. F. and Meedeniya, D. (2012). Strongly consistent transformation of partial scenarios. SIGSOFT Software Engineering Notes (SEN), 37(4):1-8.

P3: Bowles, J. and Meedeniya, D. (2010). Formal transformation from sequence diagrams to coloured petri nets. In Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC '10), pages 216 - 225. IEEE Computer Society.

Table 1 summarises the contributions for achieving the objectives

Objective	Chapter	Contribution
Ob1	3, 4, 7	P3
Ob2	5	P3
Ob3	6	P2
Ob4	6	P1, P2
Ob5	7	Thesis
Ob6	7, 8	P1

Table 1: Overview of the research contribution

1.4 Thesis Structure

This thesis is organised as follows:

Chapter 2 (Research Background) explores software development using MDD approach and related model transformation literature that are necessary to understand the approach proposed in this thesis.

Chapter 3 (A Design Model: Sequence Diagram) explains the UML SD and IOD models and gives formal representations of these models with respective trace-based languages.

Chapter 4 (A Formal Model: Coloured Petri Net) describes CPNs and some of the variations with time, stochastic and hierarchical notions. This chapter gives the formal representations and associated languages with each model.

Chapter 5 (Model Transformation: Sequence Diagrams to Coloured Petri Nets) defines and explains the easily extendable formal rules for the transformation of main constructs and interaction fragments in a sequence diagram to a coloured Petri net.

Chapter 6 (Complex Model Transformation) defines rules for partial, incremental, parametric transformations, together with model composition and integration.

Chapter 7 (Model Transformation Correctness) formally proves the correctness of the model transformation rules.

Chapter 8 (Support for Automated Model Transformation) explains the construction of the prototype tool that implements the model transformations. Also, this chapter shows the applicability of the transformations using example-based case studies.

Chapter 9 (Discussion and Conclusion) discusses and concludes the thesis summarising the contributions of this work and suggests ideas for potential future work.

2 Research Background

Successful software system development essentially relies on its design models. Model-based software development, also known as Model Driven Development (MDD), is OMG's vision of an evolving approach for software system development [Kleppe et al., 2003]. MDD-based software development specifies rather abstract models of the desired system, and automatically transforms them into more specific models, resulting in the final system.

Models and model transformations are considered as key concepts in MDD and the transformations help to improve the quality of models [Gorton, 2006]. There are non-formal or semi-formal design models that are intuitive graphical modelling languages to design complex systems, yet are not sufficient to guarantee correct system designs. On the other hand, formal design models provide a theoretical support to verify system design models. However, they are often restrictive and complex to use. The correct transformation of non-formal models into formal models comprises the advantages of both models and will offer better designs for modern complex software development needs, than practicing the individual approaches alone. Also, software development based on formal methods enables the design models to be automatically transformed into execution models and finally to the implementation code.

This chapter provides a review of state-of-the-art techniques for software development based on a MDD approach. There is a wide range of design models and theoretical methods available in the literature to design and verify software models. With an initial overview of MDD, and different model transformation types, this chapter outlines the important literature focusing on specific research areas associated with software modelling and formal model transformation of software systems. Here, we focus on UML as a graphical modelling language and coloured Petri nets as the formal model considered

in this thesis. We then turn our attention to different model transformation approaches, in particular related work on transforming UML models into Petri nets. Next we outline the importance of correct model transformations, related work on model analysis and the challenges associated with formal modelling and verification.

2.1 Model Driven Development

Software systems are constantly increasing their complexity with the rapid growth of the modern computing technology. Theses software systems have influenced in our daily activities in a scale which makes us ever reliant on their dependability. The requirement of developing reliable software systems cost effectively has been the prime motive for the model-based research initiative and associated theories. As a result, model-based software development, also known as Model Driven Development (MDD), emerged in early 2000s as an evolving approach for software system development.

A Model is a physical or abstract representation of a referent. Since, we are interested in software systems, the referent in this case is a software system. The model captures details of a system prior to development. Software system models describe different aspects of a system including the structure of the computer system that make up the system, and the behaviour of the system [Ludewig, 2003, OMG, 2011a].

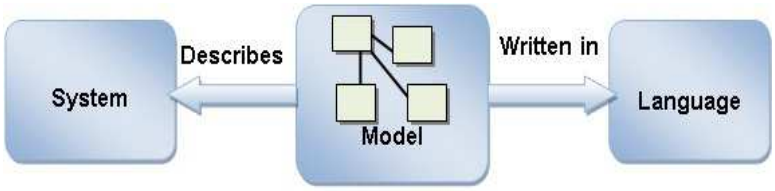


Figure 2.1: The relationship between model-system-language.

In the context of MDD, a model is a description of a system written in a well-defined language [Kleppe et al., 2003]. A well-defined language has a precise syntax that represents the elements and relationships in a model, and semantics that specifies the meaning of those elements [Douglass, 2004]. Figure 2.1 shows the relationship between a model, the system it describes and the language that is used to specify the model.

Modelling and abstraction are important aspects in software engineering when developing complex software systems. Abstraction removes unnecessary details from a model to simplify and focus the attention to general concepts that are important for the construction of appropriate models. The intention of the MDD approach is to increase the level of abstraction in system specification through models and increase automation in software development while maintaining their consistency and completeness.

In general, a single model is not sufficient to have a complete description or understanding of a system. Indeed, it is the combination of various models with different views that gives a complete system specification [Kleppe et al., 2003]. For example, Unified Modelling Language (UML), a popular graphical modelling language, contains a series of diagrams and notation to capture structural aspects (class diagram, component diagram, deployment diagram, etc.) and behavioural aspects (interaction overview diagrams, sequence diagrams and communication diagrams, activity diagrams, state diagrams, etc.) [Arlow and Neustadt, 2005, OMG, 2011a, Douglass, 2004]. Chapter 3 discusses more on UML.

MDD-based software development supports system design through a set of models that represent different system views, possibly at different levels of abstraction. This facilitates the development of correct and well-functioning software systems [Vale and Hammoudi, 2009]. Therefore, accommodating var-

ious models as required should be a vital step in the development process of systems and a way of managing complexity and quality [de Lara and Guerra, 2005, Koch, 2006]. Also, modelling with different levels of abstraction is useful between system architects and developers to clarify system structure and behaviour.

MDD-based software development facilitates automatic transformation and integration of system design models, while preserving their traceability, completeness and consistency [Kleppe et al., 2003]. Model transformations map a source model to a target model to fine-tune the constructed model into a more precise model, enable possible analysis or to make it closer to the target platform. (see Section 2.2 for more details). This approach has a positive influence on the reliability and efficiency of the software development process and has recently gained more attention from practitioners and academics in the software engineering field.

In software development, it is important to model a software system prior to the implementation for many reasons. A graphical representation of a system provides an easily understandable view of the system, which facilitates communication with the stakeholders and reduces possible misunderstandings of system requirements. Hence, a system model clarifies system functional and non-functional requirements to customers and system users.

Moreover, system design models enables the early identification of incompleteness, ambiguities, and inconsistencies in the system specification through model verification techniques [C.Baier and J.Katoen, 2008]. MDD-based software development comprises formal models and techniques that verify software design models. Formal methods, which we discuss in more detail in Section 2.4, use mathematical approaches for modelling and formal analysis. Model verification with formal methods guarantees the correctness and consistency of

the system before the actual system implementation [C.Baier and J.Katoen, 2008, Katoen, 2008, Grumberg and Long, 1991]. This helps to reduce the associated development time and cost later on, hence increase the developer efficiency and productivity [MSDN-library,].

Further, the software design models that are independent of the implementation specific details can be reused and transformed into different implementations as necessary by adding language specific details [Kleppe et al., 2003]. A further advantage of system models without implementation details is the possibility of sharing and reusing for similar domains by amending existing models.

2.1.1 Terminology and Approach

Standardisation of models and their specifications is crucial for wide acceptance and usage of models for system lifecycle activities. OMG is a globally accepted organization for defining manufacturer-independent standards, to improve the interoperability (manufacturer independence) and portability (platform independence) of software systems [Stahl et al., 2006, Kleppe et al., 2003, OMG, 2003]. In the following we outline some of the key aspects of MDD.

- Model Driven Engineering (MDE): is a software development methodology that focuses on the abstract representation of software system models, rather than on the implementation algorithm. This method supports software development by promoting communication between system users, simplifying software design and increasing system compatibility and consequently maximising productivity [OMG, 2003].

- Platform Independent Model (PIM): describes the system structure and behaviour by concealing the technological details through abstraction [de Lara and Guerra, 2005, Koch, 2006, Kleppe et al., 2003]. Hence, this

model representation approach is independent from the underlying processes, communication infrastructure, middleware, implementation language, etc. [Douglass, 2004].

- Platform Specific Model (PSM): represents a system using the concepts specific to the relevant platform where the system is being implemented. Hence, it includes both application semantics and runtime behaviour and can be considered as a detailed version of PIM with platform specific elements. A PSM model is usually obtained by applying model transformations to the PIM while adding technology specific data [de Lara and Guerra, 2005, Koch, 2006, Kleppe et al., 2003].

- Model Driven Architecture (MDA): is an OMG defined software architecture framework for the software development based on the MDD approach through model construction and model transformations [Kleppe et al., 2003, Arlow and Neustadt, 2005, OMG, 2011a]. MDA refers to the architecture of the various standards and model forms that serve as the technology, and not the architecture of the system being modelled. MDA separates application data from the underlying platform specific details, realises the PIMs built using OMG modelling standards and supports the automated transformation from a PIM to a PSM, where the PSM is used for the system implementation [Kleppe et al., 2003]. Figure 2.2 shows the major steps in MDA.

The main focus of MDA is the modelling of PIM and its transformation to the PSM, in such a way that, the transformations are defined once and applied to different software system developments. Consequently, MDD-based software development increases productivity, portability and interoperability of a software system, which are essential features in a modern software sys-

Figure 2.2: Major steps in the MDA.

tem [Gorton, 2006, Kleppe et al., 2003]. With the use of formal modelling, this approach ensures consistency, while enabling flexibility in model transformation and integration [Truyen, 2006]. MDA enhances the maintainability of software systems through well-defined PIMs, architectural separation of concerns and manageability of technological changes [Kleppe et al., 2003, Stahl et al., 2006]. Also, the MDD approach addresses some of the current issues in software development such as complexity, interoperability, re-configurability and adaptability by providing an abstract mechanism that separates the logical solution from the technical solution [de Lara and Guerra, 2005, Koch, 2006, OMG, 2003]. Further modelling components can be shared and reused by incorporating changes to the existing models. In particular, MDD provides mechanisms and techniques for creating software tools and the infrastructure necessary to allow for automated transformations.

2.2 Model Transformations

Model transformation plays an essential role in software development based on the MDD approach. Model transformation is specified by a set of transformation rules that are described using a model transformation language, which shows the mapping of the elements from a source model to a target model [de Lara and Guerra, 2005, OMG, 2003, Ehrig et al., 2008, Mellor et al., 2004] (see Figure 2.3).

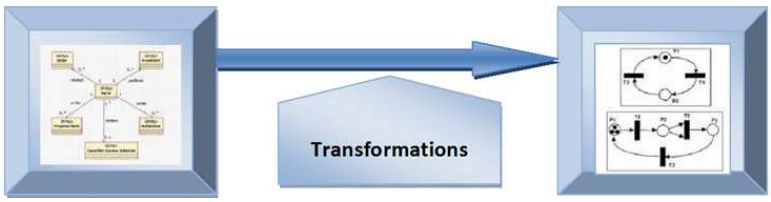


Figure 2.3: The model transformation process.

Kleppe et al. [Kleppe et al., 2003] has stated the following definition of model transformation. "A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language".

Various model transformation classifications are available in the literature [T.Mens and Grop, 2006, Mens et al., 2005, Cabot et al., 2010c, Boronat et al., 2009a, Hidaka et al., 2009] that facilitate software developers to decide the most suitable technique to use for a given model transformation process.

There are two main approaches to M2M transformations: operational and declarative. Operational M2M transformations are based on rules that explicitly describe the creation of the elements in the target model from the elements in the source model. In other words, this approach specifies the steps that are required to derive the target model from the source model by focusing on how and when the transformation has to be performed. Declarative M2M transformations by contrast are based on graphical or textual pattern for describing the relation between the source and the target model [Cabot et al., 2010c, Orejas et al., 2009]. From another point of view, model transformations can be categorised as syntactic transformation that perform the transformation between

syntactically well-formed models, and semantic transformation that maps the behaviour from source to target model [Mens et al., 2005].

Another powerful classification for M2M transformations can be expressed using endogenous and exogenous transformations [T.Mens and Grop, 2006, Boronat et al., 2009a]. The transformations between models that are expressed in the same language and different languages are considered as endogenous transformations and exogenous transformations, respectively. For example, an exogenous model transformation is used in the translation of a PIM to a PSM, where the transformation synthesises a high-level abstract model into a lower-level concrete model. A well-known example for an endogenous transformation is model refactoring that aims at improving the operational qualities of the model while preserving the semantics of the model.

Horizontal and vertical transformations are another classification of model transformations that perform on the same level of abstraction and across levels of abstractions, respectively [Mens et al., 2005]. For example, the transformation from PIM to PSM can be considered as a vertical exogenous model transformation, and the refinement of a design model can be considered as a vertical endogenous model transformation. The flattening of a composite (nested) model to a model with simple states can be considered as a horizontal endogenous model transformation, whereas the migration from one domain-specific language to another can be considered as a horizontal exogenous transformation.

Further, model transformation can be categorised as uni-directional or bidirectional [Koch, 2006, Hidaka et al., 2009]. A uni-directional transformation always takes the same type of input and produces the same type of output, whereas in a bidirectional transformation the same type of model can sometimes be the input and other times the output.

M2M transformations that support a MDD approach should guarantee some overall characteristics [Kleppe et al., 2003, T.Mens and Grop, 2006, Lano, 2009]. For example, the transformation rules should guarantee model *confluence*, that is, applying the transformation to a given source model should always generate a unique target model, in other words, the transformation is deterministic. Another important characteristic of transformations is *termination*, that is, a transformation when applied to a source model always terminates and generates a valid target model. In general, the language used to define the transformation rules should be precise, concise and clear such that the elements of the source model are clearly mapped onto elements in the target models. Moreover, the transformations should be defined in such a way that it is easy to add new rules. Finally, rules should be executable and implementable in an efficient way. By defining transformation rules formally, the properties associated with the model transformation can be obtained more directly using available formal analysis techniques [Mallet et al., 2006, Emadi and Shams, 2009a, Emadi, 2010, Ameedeen et al., 2009, Merseguer and Campos, 2004, Campos and Merseguer, 2006].

2.3 Software Design Models

Modern software systems need to function with great reliability, as software has become critical to advancement in many areas of human endeavour. Software systems can be large-scale with complex layers of control such as air traffic control systems, telecommunication systems or can be small scale and simple, such as a pocket calculator, a mobile device, etc. These systems are used in various application domains such as healthcare patients control systems [Aburub et al., 2007], real time embedded systems (elevator systems) [Fernandes et al., 2007, Radjenovic and Paige, 2010] and computer system networks (cloud

computing) [web services, ,Microsoft,]. The development of these software systems require (or at least benefit from) strong modelling and analysis methods including quantitative modelling and formal verification.

Modelling a software system is a core area in a software development process. Software design models can be mainly categorised into two types: graphical design models or formal design models (see Section 2.4 for more details). A graphical design model represents a system using diagrammatic notations. For example, the Unified Modelling Language (UML) is an industrially well-known standard, but mostly an informal graphical modelling language for the design of software systems. Live Sequence Charts (LSC) [Harel et al., 2005, D. Harel, 2003, Harel and Kugler, 2002], and Message Sequence charts (MSC) [ITU, 1999, Alur et al., 2003, Uchitel and Kramer, 2001] can be considered as other popular design models with scenario-based descriptions. Scenario or interaction is the observable behaviour of information exchange between participating entities that perform a task. In this section, we describe UML as a graphical design model.

UML is a widely used object-oriented modelling language in present software development. UML has been standardised by the Object Management Group (OMG) [OMG, 2011a] and incorporates the best practises in modelling techniques and software engineering. UML modelling can be applied to many systems in a variety of application domains varying from simple standalone applications to global enterprise solutions [Bernardi et al., 2002, Gherbi and Khendek, 2006, Tang et al., 2010, Dinh-Trong et al., 2006, Tran et al., 2006, Haugen et al., 2006, Anda et al., 2009, Haugen et al., 2005, Campos and Merseguer, 2006]. Hence, UML is a general purpose language for system modelling.

This thesis considers UML 2, which was released in 2005. UML 2 facilitates the design of complete system models with the use of new graphical syntax

compared to previous UML 1.x versions [Arlow and Neustadt, 2005, OMG, 2011a, Douglass, 2004]. For example, UML 2 contains notations that support abstraction and real-time features in a system model. Also, UML 2 is featured with nested classifiers, which are a powerful concept in software modelling that allows one to model complex behaviours.

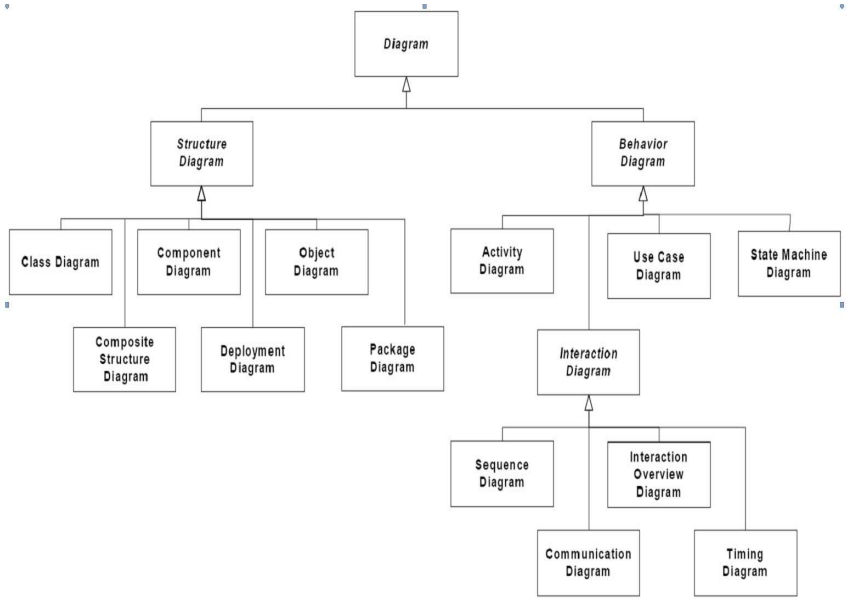


Figure 2.4: The structure of UML diagrams (adapted from [Arlow and Neustadt, 2005]).

A UML model can consist of many diagrams of different types, where each diagram presents a different view of the system. Figure 2.6 shows the organisation of UML diagrams that can be used to model structural and behavioural aspects of a software system [Arlow and Neustadt, 2005, OMG, 2011a, Douglass, 2004].

Structure diagrams such as class, object, component, deployment, and package diagrams depict a structural view of the system including concepts and properties. Behavioural diagrams such as activity, use case, state machine and interaction diagrams depict a behavioural views of a system including

methods, collaborations, activities, and state histories. Interaction diagrams can be further categorised into sequence diagrams, communication diagrams, interaction overview diagrams and timing diagrams. We consider sequence diagrams as the main design model in this thesis and Chapter 3 describes this in more detail.

The expressive power of UML 2 models can be enhanced using the constructs of the Object Constraint Language (OCL) [Arlow and Neustadt, 2005, OMG, 2006]. OCL is a widely used constraint language that specifies extra information on UML models [Cabot et al., 2008, Cabot et al., 2010b, Cavarra and Filipe, 2004]. These OCL constructs can be directly associated with UML elements as tagged values or notes. Since OCL is a passive and pure specification constraint language, the OCL constructs do not affect the UML model by changing any value.

Even though the intuitive notations of UML diagrams greatly improve the communication among developers, the lack of a formal semantics makes it difficult to automate analysis and verification of the software design models. Generally, the UML standard [OMG, 2011a] has focused on defining the syntax of models specifying the valid combination of model elements that are based on meta-models. The semantics that defines the mapping of the model language elements into a domain of values has only been defined informally. i.e. UML models cannot be used directly for formal model analysis and verification of design models.

Much work has been done or proposed for representing UML semantics in a formal way in order to enable model validation, model checking and consistency checking of design models [Harel and Maoz, 2007, Cimatti et al., 2011, Kong et al., 2009, Straeten et al., 2007, Zhang et al., 2009, Li et al., 2004, Störrle, 2004, Shen et al., 2008a, Lund and Stølen, 2006, Dan et al., 2007, Shen et al.,

2008b].

There is a range of approaches for defining semantics of UML models. Formal representation using denotational semantics is a well-established approach that maps every syntactic construct to a semantic construct of the model [Harel and Maoz, 2007,Lano, 2009,Störrle, 2004,Cengarle et al., 2006,Hammal, 2006]. For example, the work done in [Störrle, 2003a, Halvorsen et al., 2007, Haugen et al., 2006,Haugen et al., 2005,Cengarle and Knapp, 2004] has described denotational trace semantics in order to capture the meaning of sequence diagrams with time information.

Other widely used formal representations are based on operational [Lund and Stølen, 2006, Zhang et al., 2009], transformational [Kong et al., 2009], algebraic, axiomatic, and meta-modelling approaches [Lano, 2009]. Algebraic approaches map the language constructs into a mathematical algebra and meta-modelling approach uses a subset of UML itself as a semantic domain for UML.

Axiomatic semantics defines an interpretation of UML into a mathematical formalism such as first-order set theory. i.e. this technique maps language constructs into logical theories, consisting of mathematical structures together with axioms defining their properties. For example, the work done in [Cimatti et al., 2011] describes the formal representation of class diagrams, and combines fragments of first order logic (to describe rich data and relationships between attributes and entities) with temporal operators (to describe the evolution of the scenarios). A partial order semantics for UML interaction diagrams is presented in [Störrle, 2003b] and an automata theoretic semantics for scenario-based descriptions of reactive systems is presented in [Grosu and Smolka, 2005, Moschoyiannis et al., 2009]. Moreover, logic based semantics for UML interactions is defined in [Bowles, 2006, Störrle, 2003a, Runde et al.,

2005].

Operational approaches map a language into structures of an abstract execution environment. For example, work done in [Zhang et al., 2009] has defined an operational semantics for real-time state-charts, and work done in [Li et al., 2004] has presented a formal semantics in abstract syntax form to check the consistencies among different UML diagrams.

Transformational approaches map a language into another language, which already has semantics, in order to assign semantics to the source language. For example, behavioural semantics for statechart diagrams has been specified using graph transformation techniques in [Kong et al., 2009], and sequence diagrams have been formalised using Abstract State Machines (ASMs) in [Cavarra and Küster-Filipe, 2004].

Different approaches to formal representations have unique advantages and disadvantages and support different forms of analysis. For example, algebraic approaches are particularly good for reasoning about the equality of models. Axiomatic approaches support general reasoning and a comprehensive expression of language features, but at the cost of using elaborate formalisms for which the support tools may not exist. Meta-modelling and transformation approaches require the existence of a language with a well-defined semantics (for example Petri-nets). Section 2.4 describe this in more detail.

Instead of relying on basic mathematics, related work often have proposed the use of specialised formalisms such as Z [Spivey, 1992], VDM [Jones, 1990], B-specification [Idani and Ledru, 2006], Event-B [Mosbahi et al., 2011] and Object-Z [Derrick and Wehrheim, 2010] and Template semantics [Shen et al., 2008a].

In this thesis, we avoid the use of more specialised notations when defining the formal representation of UML sequence diagrams (Chapter 3), as these

notations are general and not adequate to express all concepts in UML. In particular, some of these formal notations have preferences such as use with theorem provers, constraint solvers (Alloy analyser [Nimiya et al., 2010]), time automata (UPPALL [Firley et al., 1999]) and model checkers (SPIN model checker [Holzmann, 1997, Amstel et al., 2007, Limaa et al., 2009, Yatake and Aoki, 2010]) when it comes to model analysis. For example, some of these techniques lack MDD-based high-level software concepts such as abstraction and not sufficient for object-oriented modelling [Spivey, 1992, Jones, 1990]. Further, some are capable of modelling and analysis of functional requirements [Nimiya et al., 2010,Anastasakis et al., 2010]or structural behaviour or untimed [Limaa et al., 2009] or timed behaviour [Firley et al., 1999], only. We kept the design model free of this bias to ensure that we obtain a true syntax and semantics which can be used for formal model transformation that enables future formal verifications. For these reasons we use only mathematics when formalising the design models considered in this thesis.

2.4 Formal Models

Formal models are a collection of well-defined mathematically-based techniques. Their theoretical support makes it possible to verify system designs. A complete definition of a formal modelling language consists of a description of its well-defined syntax and semantics that enhance the readability and the expressiveness of the language.

There is a growing acceptance that formal methods form an essential part of the design of any reliable software system [Ribeiro et al., ,Bowles and Bordbar, 2007,Milner, 2009,Hillston and Kloul, 2006,Cimatti et al., 2011,Mosbahi et al., 2011,Moschoyiannis et al., 2005,Benmerzoug et al., 2008,Y.Yang et al., 2005,Jensen et al., 2007]. This is because formal methods have the potential to

eliminate ambiguities, and design faults and thereby avoid the associated system failures. In particular, formal model of a system can be used to prove system properties such as performance, reachability, consistency and correctness, mathematically [Gilmore et al., 2003a, Bowles and Kloul, 2010, Hillston and Kloul, 2006, Hinton et al., 2006, Kwiatkowska et al., 2007, Katoen, 2008, Grumberg and Long, 1991]. Moreover, formal models and methods make software designs more tangible by allowing rigorous validation and verification [Silva and dos Santos, 2004, Jensen et al., 2007, Jensen and Kristensen, 2009, Cabot et al., 2008, de Alfaro et al., 1998, Rafe et al., 2009]. Validation provides assurance that the design specifies the right system, whereas verification assures the end system satisfies the specification.

General-purpose formal methods such as Z [Spivey, 1992], and VDM [Jones, 1990] were introduced before the advent of object-oriented modelling. As a consequence, they do not explicitly consider a semantic notion of object-orientation or other MDD-based high-level software concepts such as abstraction. Even though there are object-oriented extensions of such as Object-Z they are not sufficient for behavioural modelling [Derrick and Wehrheim, 2010].

In order to satisfy modern requirements of distributed and concurrent software systems, various formalisms have been developed for modelling and verification of such system [Radjenovic and Paige, 2010, Fernández et al., 2011, Dang et al., 2010, Buyya et al., 2009]. A variety of formal models including Event structures [Winskel and Nielsen, 1995, Bowles and Bordbar, 2007, Winskel and Saunders-Evans, 2007, Moschoyiannis et al., 2010], Bi-graph [Milner, 2009], Petri-nets [Petri, 1962, M. Nielsen, 1980, Orejas et al., 2010, Benmerzoug et al., 2008], PEPA [Hillston and Kloul, 2006], PEPA-nets [Kloul and Kuster-Filipe, 2006, Gilmore et al., 2003b, Gilmore et al., 2003a, Bowles and Kloul, 2010] among others are used in software development process.

Event structures model a system scenario as a set of event occurrences together with binary relations for expressing causal dependency of events (called causality) or pair of events that are exclude from occurring in the same execution (conflict) [Winskel and Nielsen, 1995, Bowles and Bordbar, 2007, Winskel and Saunders-Evans, 2007]. Other relations can be derived from causality and conflict, namely concurrency (any pair of events not related by causality or conflict is necessarily concurrent). Many variations of event structures have been defined essentially defining different kinds of relations between events. Prime event structures can be used to provide a semantics to Petri nets and can be understood as the unfolding of a net in this case.

A Bigraph is a mathematical structure consisting of two graphs, a place graph and a link graph, intended for modelling applications such as distributed and mobile systems. The main idea of bigraphs is to treat the placing and the linking of their nodes as independently as possible. Bigraphs have evolved from process calculi and are based on standard notions in graph theory [Milner, 2009].

PEPA (Performance Evaluation Process Algebra) is an extension of the well-known process algebra CCS with stochastic aspects to be able to capture performance [Hillston and Kloul, 2006]. In order to address mobile systems PEPA nets [Kloul and Kuster-Filipe, 2006,Gilmore et al., 2003b,Gilmore et al., 2003a, Bowles and Kloul, 2010] were introduced as a combination of coloured Petri nets [Vicario et al., 2009] and the stochastic process algebra formalism PEPA.

Among these, Petri nets have been widely adopted as behavioural models, because of their powerful representation capabilities, relatively cheap solution techniques and model verification capabilities [Murata, 1989, Vanit-Anunchai, 2010, Christensen and Petrucci, 2000]. Also there are a variety of Petri nets

with the ability of extensions [Billington, 2004] and integration with available tools [Kounev and Dutz, 2007, TU-Eindhoven, , Jensen et al., 2007, Delatour and Lamotte, 2003, Kounev et al., 2010] that evaluate system properties. This section, describes Petri nets, in particular coloured Petri nets in detail, which is the main formal model used in this thesis.

Petri-nets are a well-established set of formal models used by many researchers [Murata, 1989, Uzam et al., 2009, Vanit-Anunchai, 2010, Christensen and Petrucci, 2000, Benmerzoug et al., 2008, Bernardi et al., 2002, Kounev et al., 2006, Hamadi and Benatallah, 2003]. The origin of the Petri-net concept comes from Carl Adams Petri's dissertation in 1962 [Petri, 1962]. A Petri-net is a directed, connected, bipartite graph, where each node is a place or a transition. A transition is enabled, when there is at least one token in each place connected to a transition. An enabled transition can fire removing one token from each input place, and depositing one token in each output place [Murata, 1989, Bobbio, 1990].

Different types of high-level Petri-nets are available to model the event flow and object flow of diverse behaviours including asynchronous, concurrent, hierarchical, stochastic and real-time aspects [Murata, 1989, Billington, 2004, Thomas et al., 1996]. A High level Petri-net permits to follow the behaviour of a token in the Petri-net, so that any single token can be tracked within the PN [van der Aals, 1994, Billington, 2004]. These types include Coloured Petri-nets (CPN) [Jensen, 1981, Jensen et al., 2007, Christensen, 2002], Timed Petri-nets (TPN) [Vicario et al., 2009, Carnevali et al., 2008, van der Aalst, 1993], Stochastic Petri-nets (SPN) [Bobbio, 1990, Zimmermann, 2008, Carnevali et al., 2009, Haas, 2002], Queuing Petri-nets (QPN) [Kounev and Buchmann, 2006], Hierarchical Petri-nets (HPN) [van der Aals, 1994, Fehling, 1993, Elkoutbi and Keller, 1998] and Automation Petri-nets (APN) [Thomas et al., 1996, Uzam

et al., 2009].

We have used coloured Petri nets (CPNs) as the main synthesised formal model in this thesis. CPN is a well-known formal model rich in theory and practice [Jensen, 1990, Jensen et al., 2007, Vanit-Anunchai, 2010, Christensen and Petrucci, 2000]. CPNs are successfully used to model applications such as network protocols, security protocols, multi-agent applications, business processes, railway systems, distributed systems, and many industrial systems [Jensen, 1998,Kristensen et al., 2004,Benmerzoug et al., 2008,Vanit-Anunchai, 2010].

As described by Jensen [Jensen, 1981], CPN is a formal, graphical, and executable technique for the specification and analysis of concurrent, discrete event-based dynamic system. As a Petri net, a CPN too consists of places, transitions, arcs and coloured tokens. Places describe the state of the system, whereas the transitions describe the actions of the system. Arcs are used to connect places and transitions and states are changed when a transition fires. Tokens are used to fire a transition, and each token has a given type, also known as token colour. Thus tokens are distinguishable. We describe CPNs in Chapter 4.

CPNs are suitable for our approach of transforming object-oriented models, because the colours associated with the model can be used to distinguish between object types. Moreover, there are several well-established analysis tools for automatically verifying CPNs including their extensions of timed CPNs or stochastic CPNs [Benatallah et al., 2003,Kounev and Buchmann, 2006,Kounev et al., 2010,Vicario et al., 2009]. One such tool is CPNTools [Jensen and Kristensen, 2009] for editing, simulating and analysing CPN models.

In particular, CPNs have been extensively used in several application domains [Jensen et al., 2007, Jensen, 1997b, Jensen, 1998, Vanit-Anunchai, 2010,

Genrich and Lautenbach, 1981]. In [Jensen et al., 2007, Jensen, 1997b, Jensen, 1998], Jensen has used Petri-nets as a primitive to describe the synchronization of concurrent processes in a packet transferring protocol over an unreliable network. It discusses the Petri-net representation using automation simulation and model verifying methods such as state space and place invariant. As an extension for the above work, in [Kristensen et al., 2004], Kristensen et al. have discussed different case studies on modelling mobility and communication networks, healthcare systems and state space analysis. They have used CPN tools to model, analyse and simulate the systems.

Moreover, in [Hamadi and Benatallah, 2003], Hamadi and Benatallah have expressed a web service based system using Petri-net based algebra, specifying different types of services such as empty, sequence, parallel, etc. Furthermore, in [Silva and dos Santos, 2004], Silva and Santos have used Petri-nets as a formal model to represent system behaviour and have performed system validations using simulations. They have used Petri-nets not only as a case tool to model and analyse the system but also as a framework to express the requirements based on system use cases of a banking application.

Petri-nets can be used not only to model system behavioural aspects but also to ensure system non-functional properties such as liveliness and dead-lock avoidance [Christensen and Petrucci, 2000, Merseguer and Campos, 2004, van der Aalst, 1993, Jensen and Kristensen, 2009]. For example, in [Chrzastowski-Wachtel et al., 2003], Wachtel et al. have introduced refinement rules to avoid dead-locks in a Petri-net representation. They have proposed rules such as, parallel split followed by a parallel synchronization. Also they modelled a top-down work flow using hierarchical Petri nets for a flight ticket booking application and have used the HiWord tool [Benatallah et al., 2003] as a supporting tool.

2.5 Formal Model Transformation

Among many related research, this section aims to identify different model transformation approaches and existing research gaps in this area. For this purpose, the literature is reviewed in several point of views.

Generally, model transformation approaches can be categorised as sequential vs. concurrent approaches and algebraic vs. model-based approaches. Sequential model-based methods include specification languages such as B [Laleau and Polack, 2008, Idani and Ledru, 2006], Event-B [Mosbahi et al., 2011] and Object-Z [Derrick and Wehrheim, 2010], while concurrent model-based methods include CSP (Communicating Sequential Processes) and Petri-nets [Orejas et al., 2010, Kuster et al., 2004], etc.

Similarly, different formal model transformation techniques are available including graph transformations [de Lara and Guerra, 2005, Baresi and Pezz, 2005, Kerkouche et al., 2010, Beydeda et al., 2005, Bisztray et al., 2009, Grónmo and Móller-Pedersen, 2010], algebraic and logical [Boronat et al., 2005, Ehrig et al., 2008, Cimatti et al., 2011, Goknil et al., 2011, Baresi et al., 2011, Mosbahi et al., 2011] approaches and model transformation languages such as QVT [Stevens, 2007, Stevens, 2009, Boronat et al., 2009a, Cabot et al., 2010c] and ATL (ATLAS Transformation Language) [Cuadrado et al., 2011].

Even though many research studies have been carried out using algebraic approaches for model transformations, there are boundaries on applying these methods in practical use. For example, the lack of graphical support makes it difficult to understand by non-experts.

Several rule-based and relational-based approaches have been widely used in MDD. Graph Transformation (GT) is a rule-based approach that specifies the transformation of elements of one model to elements of another model using a set of transformation rules [Mens et al., 2005]. Graph Grammar, which is

used as GT rules, facilitates to obtain possible reachable graphs from an initial graph [Ribeiro et al.,].

Triple-Graph-Grammar (TGG) is a special type of graph transformation technique [Konigs, 2005, Ehrig et al., 2008, Orejas and Wirsing, 2009, Hermann et al., 2010, Cabot et al., 2010c]. TGG uses a source graph, target graph and a correspondence graph that records the information about the mapping between the nodes in source and target models. Generally, it is difficult to check consistency between two models when there are two unidirectional transformations in both ways. TGG supports bidirectional model transformation and consistency checking with multiple views [de Lara and Guerra, 2005].

Graph transformation is a promising approach for model transformation with reuse mechanisms. There is a tendency to combine graph transformation technology with XML and UML by the user community because of their familiarity with these languages. However, various graph transformation approaches are not always compatible [Mens et al., 2005]. Further, GT is supported by different tools that are used for model validation, however, often there are scalability issues [Baresi and Pezz, 2005].

Relational-based model transformation approaches specify changes that occur in a model due to a transformation [Kuster et al., 2004]. According to the mathematical nature of relations, they are suitable for multi-directional transformations. However, they are not executable and require additional expression languages to actually execute a relation-based transformation.

Query-View-Transformation (QVT) is a OMG defined relational model transformation technique [Stevens, 2009, Boronat et al., 2009a, Cabot et al., 2010c]. QVT considers model queries and views as special types of model transformation. QVT supports expressing a transformation so that it can be read in either direction between the two models; hence transformation con-

sistency can be ensured [Stevens, 2007]. However, there are limitations on representing every aspect of model queries, views and transformations. For example, QVT does not support transformations between textual models, as each model should confirm to meta-model standards.

Considering the practical aspects of using formalisms, some of these model transformations have used software systems represented in UML. Here, we are interested in model transformation from UML diagrams to Petri nets, which is the core area of this thesis.

Many researchers have highlighted the trade-off between the ease of use of UML and its lack of precision [Garousi, 2010, Naumenko and Wegmann, 2002]. Thus, many recent efforts have been aimed at transforming UML like scenario-based languages into formalisms such as temporal logic [Baresi et al., 2011, Anastasakis et al., 2010], event structures [Bowles, 2006], PEPA nets [Kloul and Kuster-Filipe, 2005, Bowles and Kloul, 2010], constraint languages [Cabot et al., 2008], automata [Grosu and Smolka, 2005] and Petri nets [Sgroi et al., 2004, Khadka and B.Mikolajczak, 2007, Campos and Merseguer, 2006, Emadi and Shams, 2009b]. However, the automata-based language used to capture the ordering of actions allowing sequential execution only.

In particular, there are several ways on transforming sequence diagrams (SDs) into Petri nets (PNs) [Ribeiro and Fernandes, 2006, Emadi and Shams, 2009b, Ameedeen and Bordbar, 2008, Ameedeen et al., 2011, Ouardani et al., 2006, Kessentini et al., 2010b, Alhroob et al., 2010, Fernandes et al., 2007, Merseguer and Campos, 2004, Bernardi et al., 2002, Eichner et al., 2005].

In [Bernardi et al., 2002], the authors have used sequence diagrams and state chart diagrams to represent the system functionalities and used Petri nets as a validation and performance analysis tool. However, they have considered only UML 1.x constructs and have not considered complex behaviour

for transformations.

Formal semantics for most concepts of SDs by means of Petri nets has been introduced in [Eichner et al., 2005]. The authors have shown the partial ordered and concurrent behaviour of the diagrams naturally within the Petri net. However, the transformations were shown only in a graphical representation.

An interesting work has been done in [Ameedeen and Bordbar, 2008,Ameedeen et al., 2011] to transform UML 2 sequence diagrams (SDs) into free choice Petri nets. They proposed that the transformation process should start by decomposing a SD into blocks and mapping them into Petri net blocks, each with a placeholder in which another Petri net block can be substituted. It has defined the transformations in a diagrammatic way considering only the event flow of the system. However, they have not considered data flow of the system. They have proved the correctness of transformation using labelled event structure as a common semantic domain to capture an identical behaviour in two models and have performed analysis for liveness, boundedness and reachability. Further, in [Ameedeen et al., 2011], they have extended their model with timing properties that allow performance analysis. However, this work has been done with conventional Petri-nets and timed Petri nets model and only the event flow of the system is considered and unable to handle object flow. In our approach we address this limitation by using CPNs.

Moreover, in [Ameedeen et al., 2009], the authors have extended their work to facilitate transformation of SDs with time aspects to semantically equivalent PN that preserves the time constraints. They have used the generated timed PN to analyse performances such as execution time computation and throughput analysis. The proposed approach has been evaluated with a Personal Area Network application. Moreover, a similar approach to transform a SD to a PN has been presented in [Alhroob et al., 2010].

Ribeiro and Fernandes in [Ribeiro and Fernandes, 2006] have presented another approach to transform SDs into coloured Petri nets (CPNs). The aim of this work is to construct animation-based CPN that reproduce the expected scenarios and thus validate them. They have considered the transformation of different interaction fragments and used a case study to validate the transformations. However, the defined semantics does not handle the object-oriented features and have represented only the diagrammatic transformation.

The work done in [Campos and Merseguer, 2006] emphasises the need of performance analysis in design stage. The authors have used UML models to gain the annotated design for time aspects and transformed them into stochastic Petri nets. Considering a basic mail client system as a case study they have analysed the model for properties such as execution times, rates and throughput. They have used ArgoSPE tool for the analysis, however, some techniques are more time consuming and require human intervention and expertise.

Most of the research in transforming UML to Petri-nets have not utilised all the structures associated with UML models [Kessentini et al., 2010b,Emadi and Shams, 2009b,Emadi and Shams, 2009a,Ouardani et al., 2006]. For example, the work done in [Emadi and Shams, 2009b,Emadi and Shams, 2009a] considers only simple structures without a formal description, when transforming UML models into Petri-nets. Moreover, a meta-model-based transformation approach from a SD to a PN has been presented in [Ouardani et al., 2006]. The transformation is limited for the representation of basic constructs including inter objects communication, hence it is not suitable for our work, as it is. Further, in [Fernandes et al., 2007] Fernandes et al. have given a non-formal description that facilitates to transform UML models into CPNs using a case study on the specification of an elevator controller.

2.6 Model Transformation Correctness

Correctness of model transformation is essential for the success of MDD approach. In model transformation, each model can incorporate with details that are not reflected in the other. In particular, when transforming UML models into mathematical domains, the results of a formal validation can be invalidated by erroneous model transformations as the system engineers cannot distinguish whether an error is in the design or in the transformation, itself.

Generally, the correctness of model transformation concerns properties such as consistency (a synthesised model is inconsistent if there are contradictions present in the source model) and completeness (a model synthesised is incomplete if there are missing elements of the source model). Moreover, the correctness of model transformations have defined in several notions such as syntactic and semantic correctness [Lano, 2009,Ehrig and Ermel, 2008,T.Mens and Grop, 2006]. Syntactical correctness ensures that the transformation always produces syntactically well-formed target model from valid source models. Semantic correctness guarantees that the target model satisfies the behavioural properties that should be preserved in the source model.

There is not much research available in establishing semantic correctness of transformations particularly for exogenous transformations [Hülsbusch et al., 2010b,Hülsbusch et al., 2010a,T.Mens and Grop, 2006]. Semantic correction is crucial for the transformation of behavioural models [Christensen and Petrucci, 2000, Lakos and Petrucci, 2004, Grumberg and Long, 1991]. However, related literature reports that the semantic correctness of the model transformations is hard to prove [Orejas and Wirsing, 2009, Greenyer and Kindlev, 2007], in the case of exogenous transformations, of having to deal with different kinds of models.. By contrast, several studies have proposed for proving the syntactical correctness of M2M transformations that have defined in a declarative way

[Schürr, 1995, Cabot et al., 2010c, Hülsbusch et al., 2010b, Orejas et al., 2009, Orejas and Wirsing, 2009, Ehrig and Ermel, 2008, Greenyer and Kindlev, 2007, Dang et al., 2010].

Most of the approaches that check for model transformation correctness are based on graph transformation techniques. For example, in [Ehrig and Ermel, 2008], the authors have specified the graph transformation rules between the models to prove the semantic correctness and completeness of the rule transformations. They have used simulation rules to define the operational behaviours of the model and have considered a case study where the target model is a Petri-net.

Another approach to verify declarative model transformations based on TGG is presented in [Cabot et al., 2010c]. They have considered UML class diagrams and associated OCL invariants, which state the conditions that must hold between models to satisfy the transformation. They have compared the expected outcome of the transformation with several scenarios to show the correctness of transformations.

Moreover, in [Orejas and Wirsing, 2009], the authors have presented an approach to proving the correctness of transformations using some general patterns that describe a given transformation and a property. In [Boronat et al., 2009a], Boronat et al. have defined algebraic specifications for meta-models. A rewriting logic based system (called Maude) was used to verify the reachability and model checking of the model. Although they approach the verification problem, they have not been concerned with whether a given transformation is correct with respect to the syntax and semantics of the models.

There are approaches that prove the semantic equivalence between the source and target models using bisimulation [Tarasyuk, 1998]. The work done by Karsai et al. [Karsai and Narayanan, 2008] has shown by finding bisim-

ulation that a synthesised model preserves the semantics of an instance of the source model with respect to a particular property (reachability in this case). However, this technique does not prove the correctness of the model transformation rules in general.

Further, in [Kessentini et al., 2010a, Kessentini et al., 2010b], the authors have aimed at transformation testing. They used a biological immune system for their validation process to detect transformation errors. They have presented meta-models for SDs and CPNs and have manipulated test cases based on each element. However, these meta-models lack elements related to formal representation and addresses only simple structures of a SD.

Most of the correctness proofs have considered only a general pattern that describes a given transformation or a property. However, analogously to syntactic and semantic correctness proofs, it is necessary to have a more general concept for showing correctness and completeness of a model transformation, independent of concrete source models.

2.7 Model Analysis

In any software system it is significant to verify a system model to reveal how it performs. Generally, formal verification of a model can be done by different techniques such as model analysis, automata theory, and simulation. Different model analysis techniques can be applied for careful monitoring of the system behaviour, identifying unreachable states and measuring properties such as liveness, reliability and performance [Mallet et al., 2006, Emadi and Shams, 2009a, Emadi, 2010, Ameedeen et al., 2009, Merseguer and Campos, 2004, Campos and Merseguer, 2006]. Also this supports to guarantee the correctness of model transformations [Hermann et al., 2010]. Moreover, early identification of flaws in a system facilitates to overcome any complications faced by the

system flow and avoid unnecessary time and cost associated with erroneous situations [Merseguer and Campos, 2004].

Different formal analysis approaches are presented in the research literature [Kwiatkowska et al., 2007,Mallet et al., 2006,Emadi and Shams, 2009a,Emadi, 2010, Ameedeen et al., 2009, van der Aalst, 1993, Christensen and Petrucci, 2000,Lakos and Petrucci, 2004,Grumberg and Long, 1991,Le et al., 2010,Baier et al., 2007]. These are not extensively explored in this review, since formal verification is not within the scope of this research. We, however, briefly present some of the selected formal verification approaches and tools from literature for an overview. It may help an interesting researcher to see how our transformation work can fit in for enabling such verifications.

The work presented in [Murata, 1989,Bobbio, 1990] describes system properties such as reachability, and liveness in a Petri net model and have showed the analysis of a Petri net. In [Bobbio, 1990], they have showed the stochastic representation of a Petri net with the use of probabilities and the Markov chain. Ameedeen et al. in [Ameedeen et al., 2009] have shown an approach to analyse time properties of a Petri net. They have considered a case study to analyse the throughput using the delay associated with the model. In [Emadi and Shams, 2009b], a simulation of Petri net has been described to analyse non-functional requirements of the modelled system. For example, the number of tokens at the starting nodes is used to denote the number of instances of the components that play the corresponding role. The movement of tokens represent the dynamic behaviour of such objects.

Another notable area is the tools that support formal verification [Kounev and Dutz, 2007, TU-Eindhoven, , Delatour and Lamotte, 2003, Jensen et al., 2007, Kounev et al., 2010]. CPNTools [Jensen and Kristensen, 2009] supports to design and simulate the CPN models with a high significance [Jensen et al.,

2007, Kristensen et al., 2004]. HiWorD is another research based tool for Petri-nets, to design hierarchical work flow modelling prototype with simulation capabilities [Benatallah et al., 2003, Chrzastowski-Wachtel et al., 2003]. ExSpect [TU-Eindhoven, , van der Aals, 1994], ORIS [Vicario et al., 2009, Carnevali et al., 2009], QPME [Kounev and Buchmann, 2006, Kounev et al., 2010] and Snoopy [Heiner et al., 2007] are some other research based tools that have been developed for Petri net analysis.

Further, in [Zimmermann, 2008], an approach to evaluate performance of a system based on stochastic coloured Petri nets has been described using a tool (called TimeNet). Also, in [Mallet et al., 2006] a time Petri net analyser (called Tina) has been used to generate behavioural graphs, in which these properties can be analysed.

2.8 Challenges of using Formalisms

Although formalisms support consistent and correct software system development, barriers exist, which prevent the wide use of formal models and methods in practice. Formalisms are based on mathematical notations, related theories and proofs. When modelling a large, complex system, the available pure mathematical notations may not sufficient or may not fit well to delineate all the graphical notations and semantics of a given system representation. This may result in for the development of sufficient amount of formal definitions and rules in a knowledge base. Also, many users lack the mathematical and abstraction skills and the required knowledge that needs to understand a system represented using formal models [Abdallah et al., 2005]. Therefore formal models are often less preferred in practice.

The deficiency of clear standards and proper documentation limits the practical use of formal models in the industry. Also, there are challenges for seam-

less integration of formal methods with the existing industrial software processes. Although formal models are crucial for critical applications, there are scalability issues when applying for industrial scale applications [Selic, 2003]. Another issue is the lack of formal language based tool support that can be used in the different phases of software development [France and Rumpe, 2007].

From another perspective, software system modelling and analysis with formal models and methods may require high initial investment. However, since these techniques are often used for complex systems, the initial costs are more tolerable than detecting and resolving system flaws at the later stages [Abdallah et al., 2005].

Similarly, Software development with the MDD approach may have some challenges. Even though, the main objective of any software development process is to obtain software systems with high quality attributes [Gorton, 2006], in MDD, it is a challenge to identify the required model transformations, which improve the qualities of a model [Saeki and Kaiya, 2007].

Also, it is a challenge to hide the complexity of the synthesised formal model and tools from the software developer. Software development based on the MDD approach lacks appropriate tool support and exchange formats, which are desirable for a seamless implementation of a software system [Koch, 2006]. However, the latter involves automated feedback mechanism, which will transforms analysis results to a form that utilises concepts in the original model [France and Rumpe, 2007]. Further, ensuring the correctness of the transformation is one of the challenges of applying formal model transformation.

The existing limitations and challenges in model transformations and their correctness may reflect inadequacies in MDD-based software development. The development of progressively correct model transformation framework will help

to move closer to a better approximation of MDD vision.

2.9 Thesis Contribution Compared to Existing Work

The literature and related work provide a strong base for this research. A summary of comparisons between the thesis contributions and the existing literature is as follows.

UML standard [OMG, 2011a,Arlow and Neustadt, 2005,OMG, 2011a,Douglass, 2004] has well-defined the model constructs including real-time features, nested classifiers and complex behaviours. It focuses on defining syntax specifying the valid combination of model elements, based on meta-models. Thus, UML lacks formal semantics that defines the mapping of its elements into a domain of values. This thesis defines a formal representation for UML SD and IOD considering the syntax and semantics of these models (Chapter 3 for more details). Moreover, it defines a formal representation for the existing time and stochastic annotations as an extension of the main SD definition.

The formal model considered for this thesis, i.e. CPN, has a well-defined theory and supported tools [Jensen, 1990,Jensen et al., 2007,Jensen and Kristensen, 2009]. The definition of a CPN in this thesis deviates slightly from the original definition and is adaptable for our purpose of modelling inter-object communication. Moreover, CPN models support real-time behaviour with the notion of a time stamp attach to tokens and hierarchical structuring by introducing so-called subnets or modules [Jensen and Kristensen, 2009]. This thesis defines these aspects differently by adding labelling functions as an extension to the main CPN definition (Chapter 4 for more details). This representation fits more naturally to the object-oriented modelling and simplifies the presentation. Here, a HCPN is defined considering only the inter-model communication with referencing labelling function that complies with the SD

decomposition mechanisms.

Different approaches have been used to transform SDs into CPNs [Bernardi et al., 2002,Ribeiro and Fernandes, 2006,Ameedeen and Bordbar, 2008,Ameedeen et al., 2011, Ouardani et al., 2006, Eichner et al., 2005]. However, several of them have considered only basic SD constructs or UML 1.x constructs. Another set of researchers have focused on event flow of the system and have not considered the handling of object-oriented features. Moreover, some have presented only graphical transformation and do not define the formal rules for the mapping. Compared to above work, this thesis defines the transformation of UML 2 sequence diagram with complex behaviours directly to a CPN-based formal space using formal exogenous transformation rules.

Most of the existing work on model transformation correctness have considered only syntactical correctness based on meta-model elements and certain properties of transformations such as confluence and termination [Schürr, 1995, Cabot et al., 2010c, Hülsbusch et al., 2010b, Orejas et al., 2009, Orejas and Wirsing, 2009, Ehrig and Ermel, 2008, Greenyer and Kindlev, 2007, Dang et al., 2010]. This thesis proves both syntactic and semantic correctness of the transformations (Chapter 7 for more details).. Thus the synthesised model preserves the same behaviour as the source model and free of implied scenarios. Therefore it can be used to analyse accurately and to perform formal verification on the models.

2.10 Concluding Remarks

This chapter has explored different software system modelling and transformation approaches that are available in the literature. In particular, UML 2 as a graphical model is used to represent the structural and behavioural aspects of a system and coloured Petri-nets as a formal model facilitates con-

current, object-oriented system modelling whilst benefiting from a rich theory and practical tools.

Even though UML is a widely used model, the expressive power of UML is not sufficient for model verification capabilities. In contrast, a formal model with underlying theory enables further analysis and formal verification of system models. Much work has been done for transforming UML models into a formal representation. The adaptation of formalisms in software development allows early identification and prevention of flaws and consequently avoids unnecessary cost associated with software development. However, the factors such as high costs and need of technical expertise when using formal methods may diminish the effectiveness of such approaches in this context.

This chapter has reviewed several approaches on transformation and validation of design models. We believe, formal model transformation supports to bridge the gap between the semi-formal graphical languages that are widely used in practice and the formal representations that are restrictive to use. This research focuses on a model-centric approach and borrows the notion of formal model transformations from MDD to construct a correct model transformation framework that enables formal verification of models.

The literature described in this chapter shows the feasibility of having further research on transforming UML models into formal models, which can enable model simulation, different forms of analysis and formal verification.

3 A Design Model: Sequence Diagram

Recent developments in software systems have increased the use of graphical modelling languages for software design. Also, with the growth of often critical software systems, it is important to accurately verify and validate software design models. A formal model representation of software systems facilitates the ability to ensure that a system model complies with the specification, and is essential for the construction of correct and consistent systems. One of the main emphases of this thesis is on the formal representation of software design models and their transformations to enable different analyses including model checking or simulation. The focus of in this chapter is given to behavioural descriptions of systems and in particular system interactions.

The behaviour of an interaction focuses on the observable information exchanged between components in a system. Interactions are often used in the software design to achieve a common understanding of the overall interactions with or within the system. At the design level we use UML2 interaction diagrams that come in different variants, namely Sequence Diagrams (SD), Interaction Overview Diagrams (IOD), Communication Diagrams (CD) and Timing Diagrams [Arlow and Neustadt, 2005, Douglass, 2004, Pilone and Pitman, 2005, Lano, 2009, OMG, 2011a]. Generally, SDs and CDs have the same expressiveness, but with different focus on timeline and structure, respectively. Timing diagrams fall into a different category and can be seen as an additional notations for capturing real-time constraints. Therefore, we consider only SDs and IODs that capture the interaction between instances and the control flow between the interactions, respectively.

Although several efforts have been made on formal representations of UML 2 SDs (as described in Chapter 2), there can still be a necessity for further research to apply verification and validation techniques in the context of real-time

and complex software system applications. Therefore, formal representation of SDs that are used for formal model transformation (as seen in Chapter 5 and 6), is one of the main focuses of this thesis.

We mainly consider denotational trace based semantics of UML2 SDs. Generally, a trace is a sequence of occurrences ordered by time that corresponds to a system run. [Micskei and Waeselynck, 2010]. Trace-semantics describe the semantics of interactions (see Section 3.1.8). When formalising a model or a language we can opt for an operational or denotational semantics. An operational semantics specifies a complete set of possible executions of a model [D. Harel, 2003, Grosu and Smolka, 2005, Kong et al., 2009, Zhang et al., 2009, Li et al., 2004, Lund, 2008]. A denotational semantics by contrast formalise the meaning of a model by constructing mathematical objects [Harel and Maoz, 2007, Eichner et al., 2005, Störrle, 2004]. Generally, a formal definition of a model describes every step that can be made in the execution of the model, where the executions are in conformance with the meaning of the language as defined by a denotational semantics. Therefore, the denotational approach assigns semantics to a language by focusing on the mapping of the syntactical elements of the model with a meaningful representation. Moreover, an operational semantics is often easily understandable to tool developers [Goknil et al., 2011, Boronat et al., 2009a] and denotational semantics is used for formal processing of the model [Eichner et al., 2005]. We use a denotational semantics based on traces of event occurrences for the formal representation of SDs.

Our description of SDs is complete and assumes all usual and special features. The latter includes incomplete messages, process creation and termination, complex behaviours such as interaction fragments and interaction uses, and extensible features such as time and stochastic artefacts. This chapter starts with a detailed description of UML2 SD notations and the following

section describes its formal syntax and semantics. Based on the semantics this chapter defines the languages (set of legal traces) for a sequence diagram. Then it describes the extensions of sequence diagrams with time and stochastic annotations. Section 3.2 describes the notations and formal representation of an IOD that uses SDs as its nodes.

3.1 A Sequence Diagram

This section gives a detailed description of a SD. The semantics described in this chapter are in accordance to the standardisation of the UML2 sequence diagrams [Arlow and Neustadt, 2005,Douglass, 2004,Pilone and Pitman, 2005, Lano, 2009, OMG, 2011a].

3.1.1 Basic Notations of a Sequence Diagram

A sequence diagram represents the interaction between the objects or components in a system for a particular purpose. It can also be used to realise a use case scenario, where a scenario describes the interactions within a system. UML2 sequence diagrams have become a widely used modelling language with many supporting tools for making SD specifications.

Generally, a SD shows a set of partial ordered sequences of messages that communicate between the instances participating in the interaction, and how the interaction develops over time along with the corresponding occurrences on the lifelines. The possible flows of control throughout the interactions in a SD are described in two dimensions: the horizontal dimension represents the different instances participating in the interaction, and the vertical dimension represents time with time progressing from top to bottom. We describe its key notations in detail first.

A SD is represented within a solid-outline rectangular frame around the

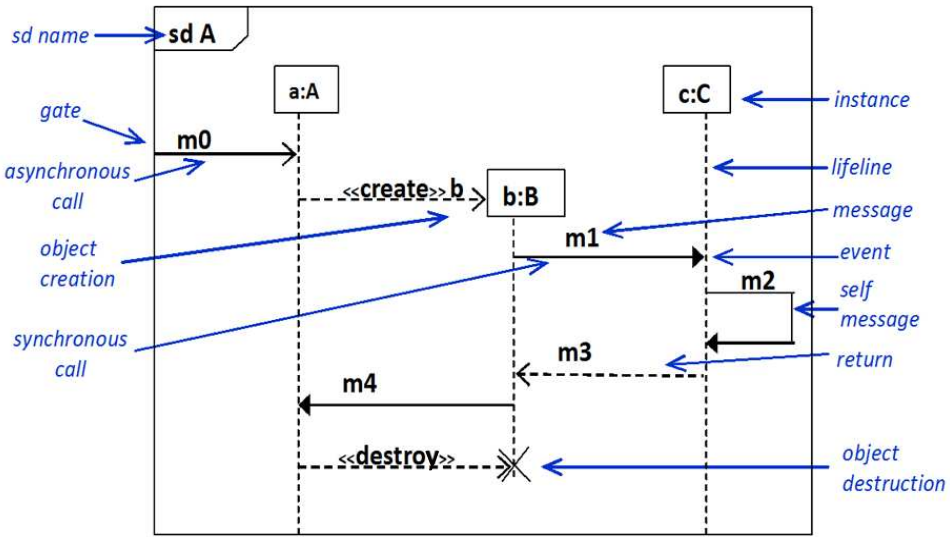


Figure 3.1: A Graphical Representation of a Sequence Diagram.

diagram that represents the boundary of the specified system. The name of the diagram following the keyword **sd** is placed inside a pentagon shaped compartment on the upper left corner of the frame. Additionally, the diagram name may followed by the input and output parameters associated with the diagram. In general, a SD shows the instances participating in an interaction. An *instance* can correspond to a particular object or a role played in an interaction. A role may be a part of a collaboration and/or an internal part of a structured class, sub-system or component. An instance has a vertical line called *lifeline* that represents the existence of the instance at a particular time. A lifeline shows the participation of an instance in an interaction. An *occurrence* is something that happens, which has some consequence within the system. The order of occurrences along a lifeline is significant for denoting the order in which these occurrences will occur. However, the absolute distance between the occurrences on the lifeline is irrelevant semantically. For instance, Figure 3.1 shows an example of a SD using basic UML 2 constructs. The SD named **A** initially contains two object instances **a:A** and **c:C**.

The most visible aspects of an interaction in a SD are the sequence of messages that are exchanged between the sending and receiving instances, along with their corresponding occurrence on the lifelines. A *message* is a named element that defines a communication between lifelines (instances) of an interaction or between a lifeline and the environment of the diagram. A message can cause, for example, an operation to be invoked, a signal to be raised, and instance to be created or destroyed. When a message represents an operation call, a message may contain the arguments of the operation, whereas in the case of a signal, the arguments of the message are the attributes of the signal. A message specifies the type of communication (synchronous or asynchronous), and the sender and receiver occurrences associated with it.

A message is represented using an arrow from the sender message end to the receiver message end. Moreover, a message with the same source and target lifeline is called as a *self-message*. In a self-message the sending message event is ordered before the receiving message event. Messages are mainly divided into two types: asynchronous and synchronous messages. In asynchronous messages, the sender sends the message and continues the execution without waiting for a return from the receiver, whereas in synchronous messages, the sender waits for the receiver to return from the execution of the message. Here, the form of the line or arrowhead reflects properties of the message [Arlow and Neustadt, 2005,OMG, 2011a]. For example, in Figure 3.1 the first message with an open arrowhead represents an asynchronous call, where the sender sends the message and continues executing without waiting for a return from the receiver. A message with a filled arrowhead represent a synchronous call, where the sender waits for the receiver to return from the executing the message. Moreover, an open arrow with a dashed line represents a return message, ($m3$ in the example) that the receiver of an earlier message returns focus of control

to the sender of that message. The message ordering, data convey via messages and associated lifelines are important in a sequence diagram, however, a SD does not focus on the manipulation of data.

Generally, when the source or target of a message is a lifeline, then it corresponds to an *event*, whereas when it is a frame, then it corresponds to a *gate*. The latter happens when the sender or receiver of the message is (locally or globally) unspecified. Gates are described in more detail later in this chapter.

During an interaction it is possible to create and destroy instances. Figure 3.1 shows an object creation and destruction messages drawn as a dashed line with an open arrowhead and the stereotype `<<create>>` and `<<destroy>>`, respectively. The creation results in creating an instance of the classifier specified by the receiver. When an object is destroyed its lifeline stops and no further occurrences are possible. Destruction is represented by a cross in the form of a `X` at the bottom of the lifeline. If it consists of a compound object it may lead to the subsequent destruction of other objects owned by composition.

Consider the example of a sequence diagram SD_A shown in Figure 3.1. The interactions within the diagram start by instance **a** receiving an asynchronous message **m0** from a gate (unknown sender). Asynchronous messages are shown using an open arrowhead and the sender continues executing without waiting for a return message. Then, instance **a** sends an object creation message to create instance **b**, where the classifier is specified by the receiver. Next instance **b** sends a synchronous message **m1** to instance **c**. Next instance **c** executes a self-message **m2**. After that instance **c** sends a return message **m3** to instance **b**, focusing the control to the sender of an earlier message **m1** and it is denoted by a dashed line. This is followed by message **m4** sent from instance **b** to instance **a**. Finally, instance **a** sends an object destroy message

to destroy instance b and the lifeline of instance **b** terminates with an X.

UML 2 SDs contain other types of messages such as lost and found messages [Arlow and Neustadt, 2005, OMG, 2011a]. A lost message is a message that will never reach its destination, and is represented using a small black circle at the arrow-end. This type of messages may be used to indicate error conditions in which messages are lost. On the other hand, for a found message, the sender is unknown or outside the scope of the interaction and is denoted by a small black circle at the starting end of the message. These messages can be used to show message from an unknown sender.

The notion of *gate* mentioned earlier can simulate the found messages or lost messages, but are more general. A gate is used to define an unspecified source or recipient of an interaction, where the corresponding lifeline of the instance is not a part of the diagram. It is considered as a syntactic interface of the SD with its environment. A gate has no symbol of its own, and simply is shown as a message pointing to/from the edge of the frame of the diagram.

A SD may also include local variables that support data flow within the interactions. These variable definitions may appear near the top of the diagram frame. Further, a SD can be structured further using complex constructs named interaction fragments and interaction uses. The remaining of this section describes these notions in detail.

3.1.2 Interaction Fragments in a Sequence Diagram

A UML 2 SD may contain constructs called *interaction* fragments denoted by a solid-outlined rectangle (see Figure 3.2). *Interaction fragments* are a way to add some more structure to part of an interaction. An interaction fragment has one *operator*, one or more *operands* and zero or more *guard conditions*, which all together help to model an interaction more clearly. Graphically, the

regions corresponding to the operands are shown by separating the interaction fragment using dashed horizontal lines [Arlow and Neustadt, 2005, Douglass, 2004, Pilone and Pitman, 2005, Lano, 2009, OMG, 2011a].

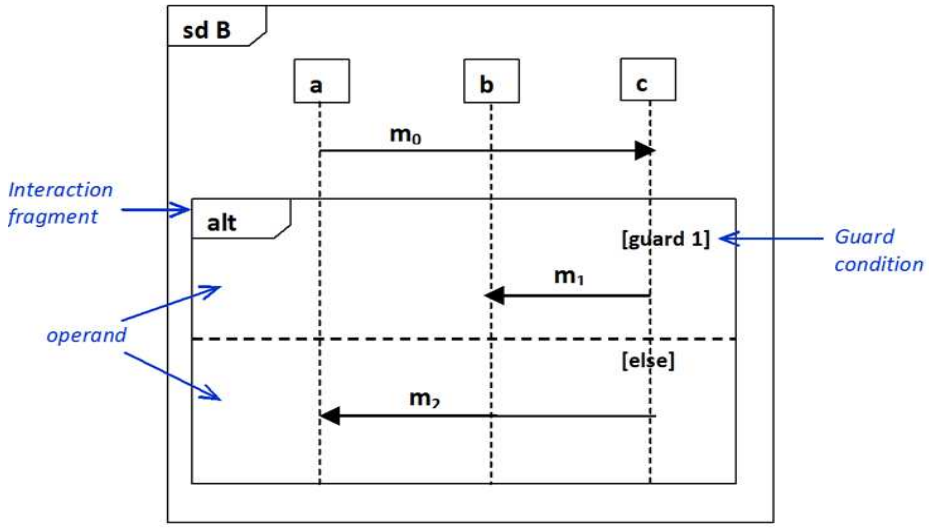


Figure 3.2: The interaction fragment behaviour of a sequence diagram.

Graphically, an operator is shown in the upper left corner of the fragment. The *operator* determines how its operands are executed and consists of one or more operands. For example, the UML standard [OMG, 2011a] defines nine unary operators: *opt* (optional behaviour), *break*, *loop* (iterative behaviour), *critical*, *neg* (forbidden behaviour), *assert* (mandatory behaviour), *ignore*, *consider* and *ref* (reference to another diagram), and the four operators *seq* (sequential behaviour), *alt* (alternative behaviour), *par* (parallel behaviour) and *strict* (strict ordering behaviour) that may be viewed as binary or n-ary. We return to these operators in more detail later.

A *guard condition* is a Boolean expression that determines whether its operand executes or not. Graphically, a guard condition is shown in square brackets covering the lifeline where the first event occurs. The values referred

in a guard may be local to the lifeline in which it resides or may be global to the whole interaction. When a condition is associated with an interaction operand, a valid set of traces can be obtained, only if the guard expression evaluates to true. Further, an operand may contain another interaction fragment as well, in the case of nested fragments.

The SD in Figure 3.2 shows an **alt** interaction fragment behaviour with two operands. The SD with name **B** begins its interaction by a message **m0** being sent from instance **a** to instance **c**. Then instance **c** makes a choice based on the guard condition, which evaluates to true and sends the message m_1 to instance **b** or message **m2** to instance **a**.

The interaction operators defined in UML 2 specification [OMG, 2011a], are capable of modelling almost every behavioural aspect of a system. Below we give an informal semantics of interaction operators associated with interaction fragments. The formal use of these operators is described in Chapter 5, when defining the transformation of each interaction fragment to the corresponding CPN.

alt : The *alt* interaction operator defines an alternative interaction fragment that represents a choice of behaviour. In this case, at most one operand is selected to execute based on the guard condition that evaluates to true at the point of the interaction. Also, the guarded operands may not lead to deterministic choice. Moreover, an operand may guarded by an *else* that represents the negation of all other guards enclosed in the interaction fragment. So that the set of traces that defines a choice is the union of the guarded traces of the operands.

opt : The *opt* interaction operator designates the option behaviour and executes only if the guard condition is true. That is it represents a choice of behaviour where either the (sole) operand happens or nothing happens.

Conceptually, options are similar to an *alt* interaction fragment with one operand.

loop : A *loop* indicates an iterative behaviour, where the contained event occurrences are to be repeated for some number of times. The loop may be infinite or have a specified number of iterations. A guard condition may include a lower and an upper number of iterations of the loop as well as a Boolean expression. A guard condition associated with the loop operand is evaluated each time at the beginning of the loop fragment, and if the guard is evaluated to true, the scenarios within the loop operand happen, otherwise the loop terminates. The loop fragment is executed as long as the guard condition is true.

break : The *break* operator represents a breaking scenario. A break is normally used in combination with a loop interaction fragment to force the exit of the loop under a certain condition or even the whole diagram depending on the context of the break itself. When the guard expression within the break operand is evaluated to true, the scenario within the break operand happens and it ignores the remainder of the enclosing interaction fragment. When the guard expression is evaluated to false, the break operand is ignored and the rest of the scenarios within the enclosing interaction fragment happen.

Figure 3.3 shows an example of a *loop* interaction fragment with a nested *break*. When the loop condition **guard 1** is evaluated to true, instance **a** sends message m_0 to instance **c** and evaluates the condition in the break interaction fragment. If **guard 2** is true, then m_1 occurs and the loop is terminated without m_2 being executed. If **guard 2** is evaluated to false, then the break fragment is ignored and continues with the rest of the interaction within the

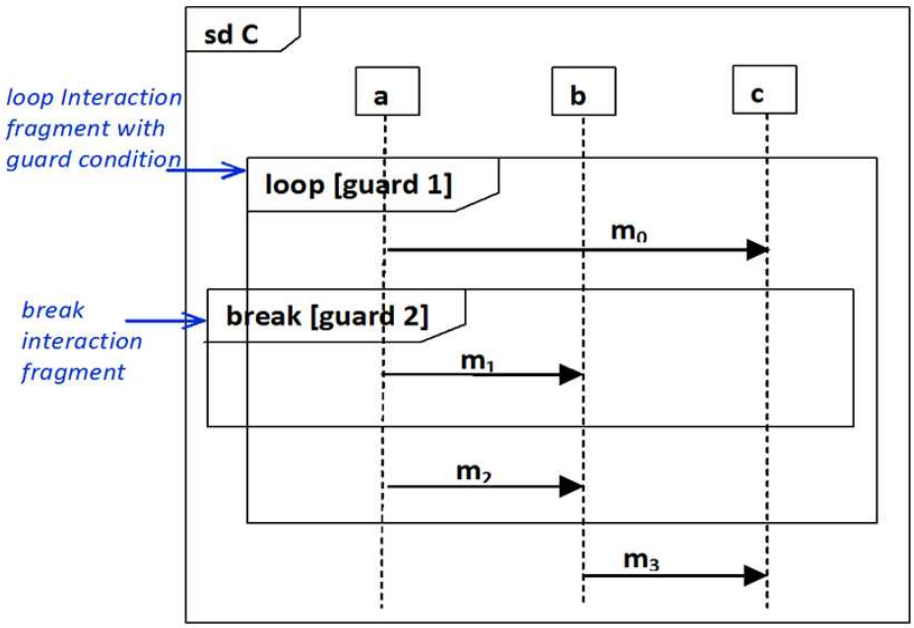


Figure 3.3: An example of a loop and break fragment combined.

loop until the loop condition, **guard 1** is false or **guard 2** is true at a later iteration. If the boundary of the break fragment is at the higher interaction level then its occurrence would lead to the termination of the entire interactions (i.e. m_3 would not occur).

par : The *par* operator defines parallel or concurrent regions in an interaction fragment. The event occurrences of different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.

critical : The interaction fragment operator *critical* represents behaviour that cannot be interleaved with other behaviours in any way. The interactions within a critical region are treated as atomic and cannot be interrupted.

Normally, a critical interaction fragment is nested within the parallel regions to ensure that a group of interactions cannot be separated. As shown in Figure 3.4, although the enclosing *par* interaction fragment implies that

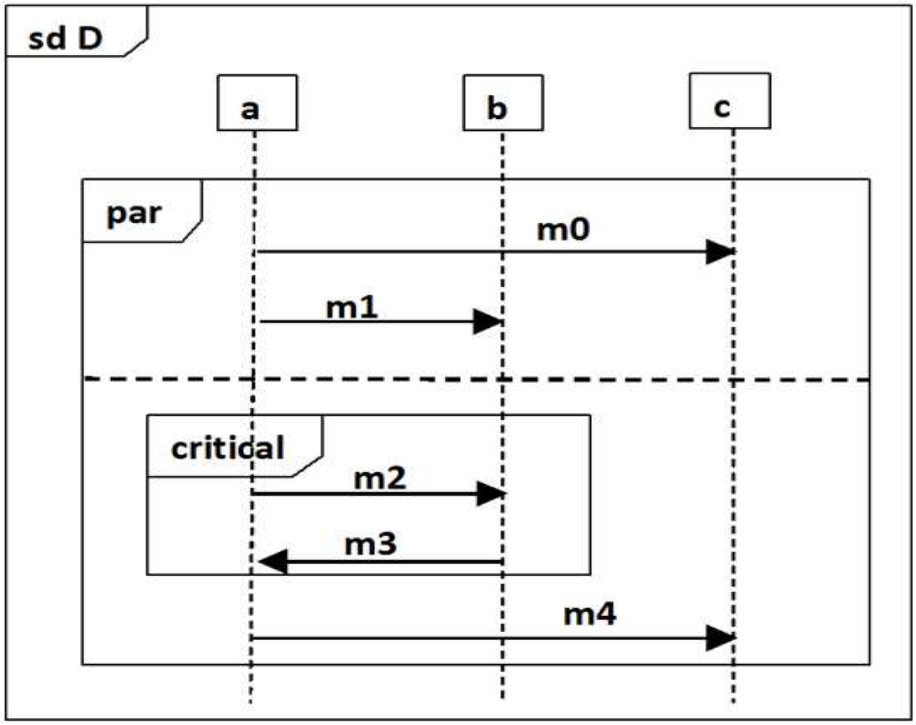


Figure 3.4: An example of a parallel and critical fragment combined.

some interaction may interleave into the region, the set of traces of enclosing constructs are restricted by the *critical* interaction fragment. For example, the valid traces are $m_0 \cdot m_1 \cdot m_2 \cdot m_3 \cdot m_4$, $m_0 \cdot m_2 \cdot m_3 \cdot m_1 \cdot m_4$, $m_0 \cdot m_2 \cdot m_3 \cdot m_4 \cdot m_1$, $m_2 \cdot m_3 \cdot m_4 \cdot m_0 \cdot m_1$, $m_2 \cdot m_3 \cdot m_0 \cdot m_4 \cdot m_1$, and $m_2 \cdot m_3 \cdot m_0 \cdot m_1 \cdot m_4$. The trace $m_0 \cdot m_2 \cdot m_1 \cdot m_3 \cdot m_4$ is an invalid trace, as the interaction with messages *m2* and m3 are in the *critical* region and are considered as atomic execution that cannot be interleaved with other interaction occurrences.

This behaviour can be described using a real-world example as shown in Figure 3.5. The SD named *callHandler* shows the handling of speed-dial calls in an interleaved manner. Since the emergency number 999-call is included in the *critical* region, it must be executed without interleaving with other calls. That is, the operator must make sure to forward the 999-call before doing

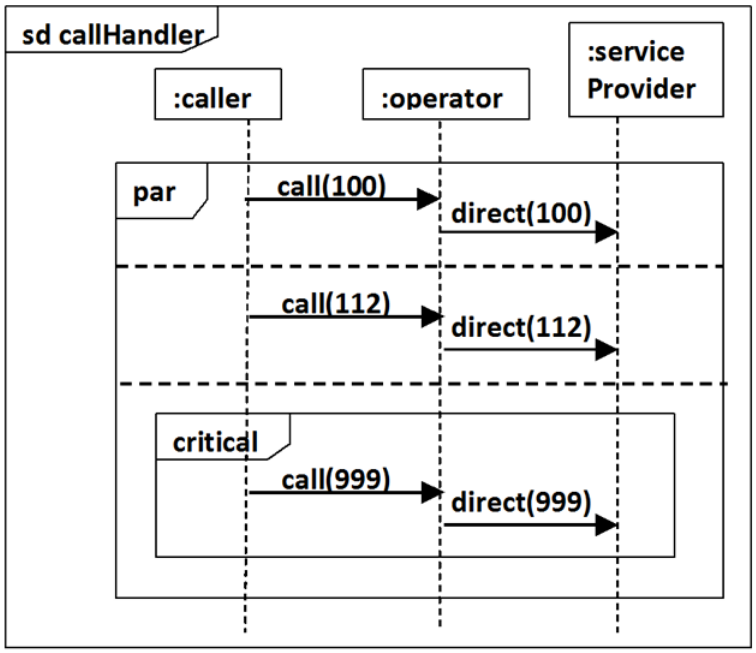


Figure 3.5: An example of a parallel and critical fragments.

anything else. The normal calls within the *par* interaction fragment, however, can be freely interleaved.

seq : The operator *seq* designates a weak sequencing between the interactions of the operands. This represents the default behaviour in a SD that preserves the occurrence order within each of the operand and on the same lifeline from different operands. This preserves the causality of messages. However, the occurrence order on different lifelines from different operands may come in any order. Thus weak sequencing reduces to a parallel merge when the operands are on disjoint sets of participants.

strict : This operator indicates strict sequence where the ordering of the interactions between operands is significant across lifelines, not just within

the same lifeline as with *seq*. The operator *strict* specifies that the messages in the interaction fragment are totally ordered.

assert : The interaction operator *assert* designates an assertion, which is a *must* behaviour. It represents the interactions that can be considered as the only valid continuations. This interaction fragments is often combined with *ignore* and *consider* operators, to indicate a compulsory behaviour at a certain point in the interaction (see below description).

neg : This operator represents interactions that are defined to be invalid or negative behaviour, meaning the interaction should be disallowed or must not execute. All interaction fragments that are different from negative are considered positive meaning that they describe interactions that are valid and should be possible.

ignore : The interaction operator *ignore* represents interactions that can be considered as insignificant and can be ignored if they appear in a corresponding execution. These interactions can be intentionally omitted from the execution. This typically implies that the interactions within the *ignore* interaction fragment are irrelevant for the purpose of the diagram, however, they may still occur during the actual execution (see example in Figure 3.6).

consider : The interaction operator *consider* represents the interactions that are explicitly relevant and should be considered within the fragment. This is equivalent to defining every other message to be ignored (see example in Figure 3.6).

The behaviour of assert, negate, ignore and consider fragments can be described using a real-world example (adapted from [Douglass, 2004]) as shown

in Figure 3.6. The SD *funcElevator* shows the functionalities related to opening, closing and moving of an elevator. When a user presses the open button in the button panel a command is triggered from the button panel to the door and the door opens.

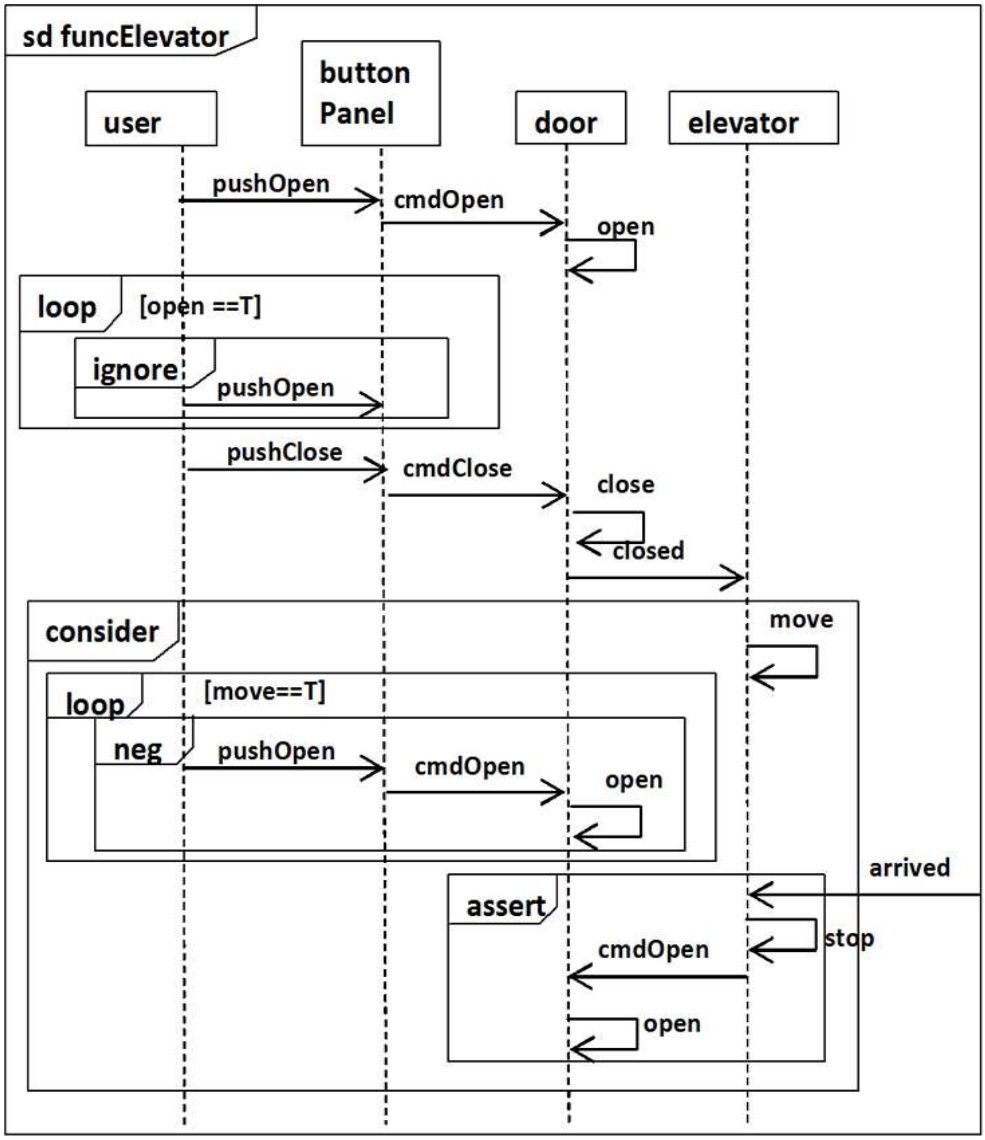


Figure 3.6: An example of a assert, negate, ignore and consider fragments.

As shown in the diagram, while the door is open, pressing the open button

(resulting in a CmdOpen message) is ignored. After a user presses the button to close door, a sequence of interactions is executed for the closing functionality of the door and the elevator starts to move. The interactions within the *consider* fragment are considered as important. As long as the elevator is moving the door cannot be opened. Finally, the *assert* operator nested within the *consider* fragment indicates that the *stop* message must follow the *arrived* message from a gate, and directs in opening the door when the elevator reaches to a floor. Here, the interaction within the *assert* ensure that the elevator must stop and open the door once it arrives at a given floor.

We can describe the difference between the behaviour within the *neg* and *ignore* fragment as follows. As described previously, the interactions enclosed in *neg* must not happen in the context and all other interactions are valid for execution. However, within the valid interaction there can be interactions that might need to be skipped from execution depending on the context. In order to achieve this behaviour UML standard uses the fragment *ignore*, where the enclosed interactions can be omitted from execution. Thus, *neg* behaviour is considered as invalid execution. The *ignore* behaviour is a valid interaction, yet we do not consider for the execution.

ref : The *ref* operator references an interaction, which appears in a different diagram. This fragment is called an *interaction use* and will describe later in this section.

When a SD becomes more complex with all these constructs, there may be a need to split the diagram and show part of the interaction on a separate diagram. Also, this helps to use part of an interaction in more than one SD. The decomposition mechanisms supported by UML 2 SDs are described as follows.

3.1.3 Decomposition in a Sequence Diagram

Decomposition facilitates the construction and understandability of complex interactions. In UML 2 it is possible to link SDs by creating references from an interaction to a separate diagram in two ways: referencing interaction fragment (called interaction uses) and lifeline decomposition.

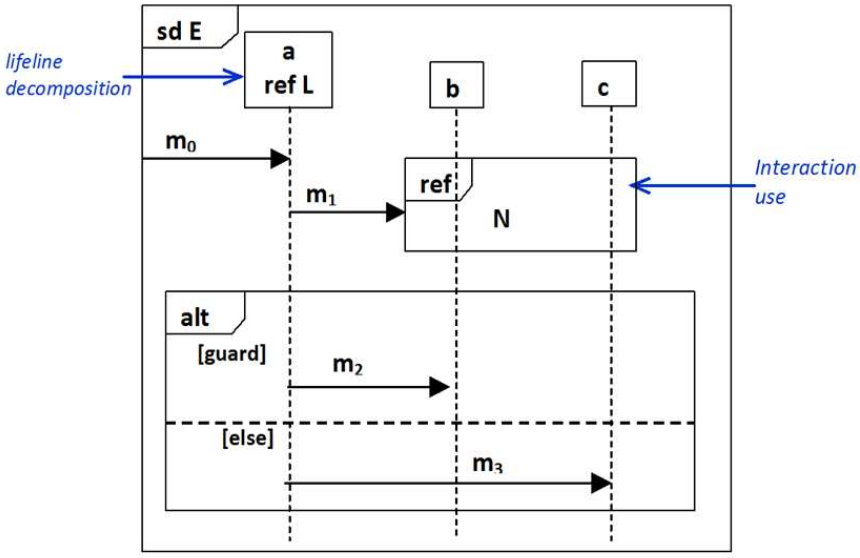


Figure 3.7: The decomposition behaviour of a sequence diagram.

Consider the example of a SD using UML 2 constructs with decomposition behaviour shown in Figure 3.7. The instance a is decomposed in another diagram named L. The instance a receives a message m_0 from a gate and sends the message m_1 to a gate in the interaction use (ref) fragment that refers a diagram named N. Then the instance a makes a choice between sending the message m_2 to the instance b or sending the message m_3 to the instance c.

Decomposition with *interaction use* (*ref* interaction fragment) refers to another SD. The reference interaction fragment helps to hide a set of interactions shown in another diagram. In this case, the referred diagram must contain all the instances covered by the *ref* interaction fragment and may contain more

instances. Here, if the referred interaction contains any incoming parameters or return values, the ref interaction fragment must also contain corresponding variables.

Lifeline decomposition indicates that an instance itself is decomposed in another diagram and is particularly useful when modelling component-based systems where the internals of a component are intentionally hidden. For example, in Figure 3.7 the instance a can be replaced by a similar or updated component, and even if its internal behaviour is quite different, the interaction in diagram E remains unchanged. The lifeline decomposition allows managing the complexity of SDs by combining several lifelines into one.

3.1.4 Additional Annotations of a Sequence Diagram

UML 2 sequence diagrams can be extended with variants such as time constraints to express real-time behaviours. For example, an interaction may incorporate time aspects that indicate the beginning or end time of an interaction occurrence (event), the duration of an interaction, and so on. UML standards use the notion of a time value (timestamp) to indicate the time associated with interactions [Arlow and Neustadt, 2005, OMG, 2005, OMG, 2011a]. The time associated with an interaction is shown using parameterised message, where the time value is assigned to the message name or an anonymous attributes of the message.

Moreover, UML provides a notation to capture a specific time associated with an event using the notion of a timer or a system clock. Graphically, a small horizontal line next to an event is placed to capture the time of the occurrence, or place a timing constraint on it. Typically, a variable is used to capture a specific instance in time and then represent constraints as offsets from that time. These time duration constraints are expressed within the curly

brackets and placed next to the message names or between event occurrences along a lifeline.

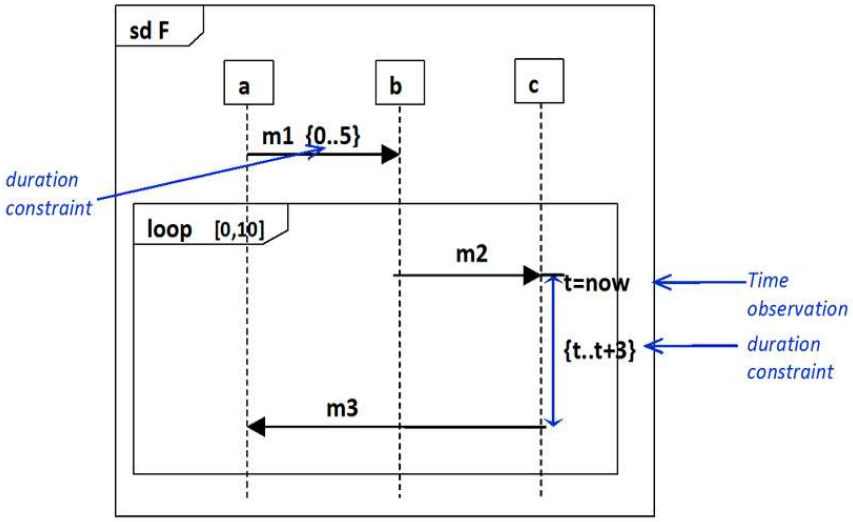


Figure 3.8: A sequence diagram with time constraints.

The SD **F** in Figure 3.8 shows the incorporation of timed constraints with the interaction occurrences. The sending and receipt of message **m1** is constrained to take between 0 and 5 time units. Further, the **loop** interaction fragment contains interaction occurrences with time constraints. As the diagram shows, the interval between receiving of **m2** and sending of **m3** there can pass at most 3 time units. Here, t is the observed time at receiving of **m2** by instance **c**.

The UML profile has not a given separate annotation to represent the stochastic aspects associate with the interactions of a SD. However, they have used the time values to derive probability related data [OMG, 2005]. We will consider the formal representation of such annotations in Section 3.1.9.

3.1.5 Formal Model of a Sequence Diagram

The informal semantics of UML2 sequence diagrams allows for many ambiguities and different interpretations of the same diagram. Therefore, when a UML2 SD is used in software system design, it is important to have a common interpretation of the language among the people who are involved in system design. Moreover, formal semantics of a model is beneficial in many ways such as to enable the comparison of specifications at different levels of abstraction and formal verification and validation of the model.

UML already partially adopts a denotational semantics to describe aspects of the language. For example, the meta-modelling approach supports the description of denotational relationships, where model elements can be abstracted as classes and their relationships can be formalised by associations [OMG, 2011a]. However, UML2 SDs lack precise formal description of semantics, when they are used in modelling of the interactions between objects, and such a formal definition would be a major amount of work. The definitions given here consider a trace based denotational semantics for sequence diagrams. Our defined semantics complies with the UML standard.

Further, when defining the formal models we have considered both *local* and *global* view of the model. A local view corresponds to an instance view of the interaction, i.e. we only consider event occurrences along the instance lifeline. By contrast a global view covers the interactions between several lifelines with the use of interaction fragments.

Consider the example shown in Figure 3.9 that highlights the notations of the formal representation given in Definition 3.1. The SD consists of elements such as name, instances, events, message labels, local transitions, interaction fragments, and associated functions (see the description later on). This diagram explicitly illustrates the events (represented by circles) and states loca-

tions (represented by ovals) that belonging to object and environment instances in a SD. In our formal representation we consider environment instances as the external instances to the system that are involved in the interaction through the presence of gates.

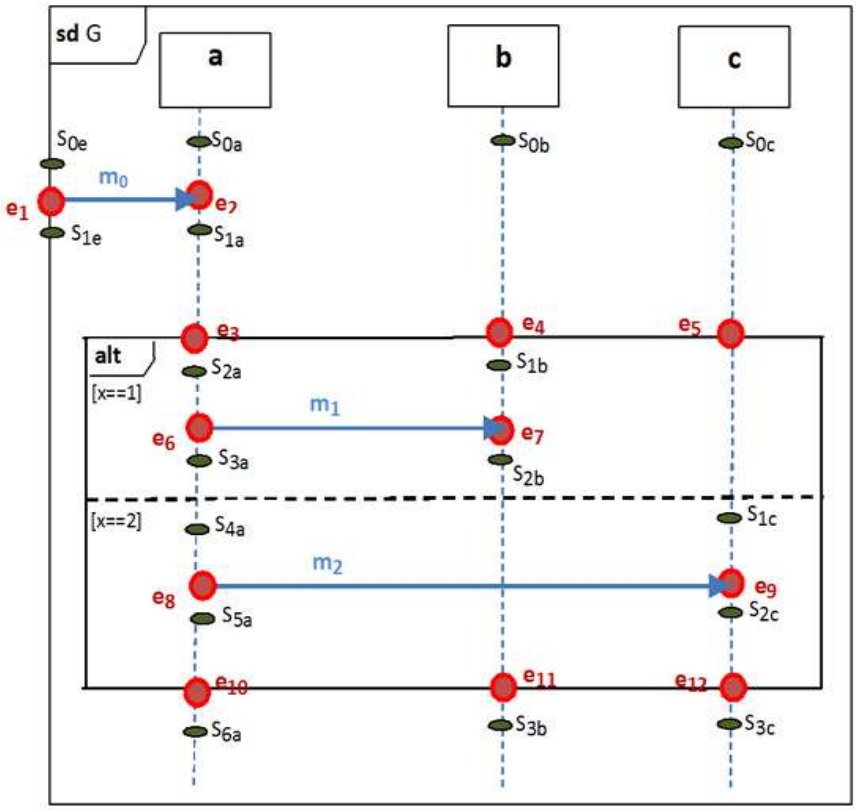


Figure 3.9: Illustrating state locations and events.

For the formal representation of sequence diagrams, we define a SD with name $d \in \mathcal{N}$ formally as a tuple SD_d, and omit d indexes from all sets when these are clear. Let Ω be a set of interaction operators given $\Omega = \{alt, par, loop, option, break, critical, assert, neg, strict, seq, consider, ignore, ref\}$, and Env be a finite set of environment instances.

We define a SD formally as follows.

Definition 3.1 (A Sequence Diagram) *A sequence diagram with name* $d \in \mathcal{N}$ *is a tuple* $SD_d = (I, E, <, M, T, F, ref, X, Exp)$ *where*

- I *is a finite set of object instances, and* $I^+ = I \cup Env$*;*
- $E = \bigcup_{i \in I^+} E_i$ *is a set of events such that for any* $i \neq j \in I^+$, $E_i \cap E_j = \emptyset$*;*
- $<$ *is a set of partial orders* $<_i \subseteq E_i \times E_i$ *with* $i \in I$*;*
- M *is a finite set of message labels;*
- T *is a set of local transitions such that* $T \subseteq E \times M \times E$ *and (1) for* $t_1, t_2 \in T$ *if* $t_1 = (e_{11}, m_1, e_{12}) \neq t_2 = (e_{21}, m_2, e_{22})$ *then* $e_{11} \neq e_{12} \neq e_{21} \neq e_{22}$*, (2) if* $t = (e_1, m, e_2) \in T$ *then* $\neg(e_1, e_2 \in E_j)$ *for* $j \in Env$*;*
- F *is the set of interaction fragment identifiers in* d *such that*
  - $f : F \to \Omega \times \mathbb{N}$ *is a function that associates an operator and a natural number (number of operands) to an interaction fragment identifier;*
  - $g : F \times \mathbb{N} \to 2^E$ *is a function that associates a set of events to a pair* (id, n) *where* id *is an interaction fragment identifier and* n *is* id*'s* n*-th operand. It is only defined if* $f(id) = (o, m)$ *and* $n \leq m$*. For arbitrary* $n \neq k$ *with* $n, k \leq m$*,* $g(id, n) \cap g(id, k) = \emptyset$*;*
  - $h : F \times \mathbb{N} \to 2^F$ *is a function that associates a set of interaction fragment identifiers to a pair* (id, n) *where* id *is an interaction fragment identifier and* n *is the* n*-th operand of the fragment. It is only defined if* $f(id) = (o, m)$*,* $o \neq ref$ *and* $n \leq m$*, and further satisfying the following properties for* $id_1, id_2 \in F$ *with* $id_1 \neq id_2$*,* $f(id_1) = (o_1, n_1)$ *and* $f(id_2) = (o_2, n_2)$*:*
    1. $id_1 \notin h(id_1, x)$ *for any* $x \leq n_1$*;*

2. *if* $id_1 \notin h(id_2, m_2)$ *and* $id_2 \notin h(id_1, m_1)$ *with arbitrary* $m_1 \leq n_1$ *and* $m_2 \leq n_2$*, then* $g(id_1, m_1) \cap g(id_2, m_2) = \emptyset$

3. *if* $id_2 \in h(id_1, m_1)$ *then* $g(id_1, m_1) \supseteq \bigcup_{n \leq n_2} g(id_2, n)$

– $j : \mathcal{N} \cup (F \times \mathbb{N}) \rightarrow 2^{I^+}$ *is a function that associates a set of instances to a diagram or to a pair* (id, n) *where* id *is an interaction fragment identifier and* n *is* id*'s* n*-th operand. It is only defined if* $f(id) = (o, m)$ *and* $n \leq m$*.*

- $ref : I \cup F \rightarrow \mathcal{N} \setminus \{d\}$ *is a partial function that associates to an object instance or an interaction fragment identifier a referenced diagram name. For arbitrary* $i \in I$ *such that* $ref(i) = n$ *for some* $n \neq d \in \mathcal{N}$ *with* $SD_n = (I_n, E_n \ldots)$*,* $E_i \subseteq E_n$ *and* $i \notin I_n$*,* ref *is only defined for* $id \in F$ *iff* $f(id) = (ref, 1)$*. Furthermore, if* $ref(id) = n$ *then* $j(id) \cap I_d \subseteq I_n$*.*

- $X = \{X_i\}_{i \in I}$ *is an* I*-indexed family of local variables;*

- Exp *is a set of expressions such that* $guard : T \rightarrow Exp$ *is a partial function that associates an expression (guard) to a local transition.*

A SD has a unique name d. The set I (or more accurately I_d) denotes the set of object instances involved in the interaction described by d whereby I^+ (or I_d^+) includes the environment instances Env. Each object instance has a lifeline and each instance $i \in I^+$ has an associated set of events E_i. For example, in Figure 3.9, there are four instances $a, b, c, v_1 \in I^+$ involved in the SD_G such that $I_G = \{a, b, c\}$ and $Env_G = \{v_1\}$.

An event describes an occurrence that has a location in time and space. That is an event is something that happens on a life of an instance at a point in time, and has no duration. For object instances, events correspond to: the sending or receiving a message, the beginning or ending of an interaction

fragment. Events associated with the environment instances are restricted to sending or receiving a message. Further, different instances do not share events. The order of the events along a lifeline is significant denoting, in general, the order in which these events will occur. The events along a lifeline (for object instances) are partially ordered, or totally ordered if the lifeline is not involved in any *alt* or *par* interaction fragments (which is described later in this section). We write $e \rightarrow e^{'}$ for *immediately following events*, that is, events with no other event in between: formally, $e <_i e^{'}$, $e \neq e^{'}$ and for all $e^{''} \in E_i$, if $e <_i e^{''} <_i e^{'}$ then $e^{''} = e$ or $e^{''} = e^{'}$. We cannot determine the ordering of events for environment instances and the partial order is therefore only defined over object instance events. When a message is sent between lifelines, the corresponding event occurrences are independent from each other. Obviously, the only constraint is that the sending of a message should occur before the receiving of that message. In Figure 3.9 events are represented from $e_1, e_2, \cdots, e_{12}$ and for a given instance the events are partially ordered such that for $a \in I$: $e_2 <_a e_3$.

A transition corresponds to a state change as a consequence of an event occurrence. We introduce the concept of local transitions denoted by set T, to represent message passing between two instances. A local transition is represented by an arrow from the sending instance to the receiving instance. A local transition $t \in T$ is a triple $(event_1, message, event_2)$ which represents an interaction between the instances associated with both events, and a self-interaction if the instances are the same. Events in local transitions are necessarily different, and for a self-transition for instance i, in particular we have $event_1 <_i event_2$. SD_G shown in Figure 3.9 consists of three local transitions $t_0, t_1, t_2 \in T$ such that $t_0 = (e_1, m_1, e_2)$, $t_1 = (e_6, m_0, e_7)$ and $t_2 = (e_8, m_2, e_9)$.

Sequence diagrams can be structured further using interaction fragments.

A particular SD may use a (finite) set of interaction fragments, and we assume that each fragment in the diagram has a unique identifier (an element in the set F). Each interaction fragment identifier $id \in F$ has an associated operator (an element in the set Ω). In order to manipulate interaction fragments as needed we define functions f, g, h, j and characterised as follows.

Function $f(id)$ returns the operator and the number of operands in the interaction fragment id. For example, as the *loop* fragments contain only one operand only, we always have $f(id) = (loop, 1)$. Similarly for the interaction fragments with the operators neg, $assert$, $consider$, $ignore$, $critical$, opt, $break$ and ref. In particular, the SD in Figure 3.9 contains an alt interaction fragment with two operands such that $x \in F$ where $f(x) = (alt, 2)$.

Function g associates a subset of events for each operand within an interaction fragment. These events cannot be shared by different operands of the same interaction fragment. We use $g : F \rightarrow 2^E$ to denote the complete set of events of an interaction fragment (or g_i if specifically for instance i). This set consists of the union of the events associated with each operand and additional events (two per instance) marking the beginning and ending of the fragment.

Formally,

$$g(x) = \bigcup_{n \in \mathbb{N}} g(x, n) \cup \bigcup_{i \in j(x)} \{e_b^i, e_e^i\}$$

where e_b^i and e_e^i denote respectively the begin and end events in fragment x for instance i. We write $\overline{g(x)}$ to denote the begin and end events only, such that, $\overline{g(x)_i} = \{e_b^i, e_e^i\}$. For example, if id is such that $f(id) = (ref, 1)$ then, only $g(id, 1)$ is defined and the reference fragments do not contain events other than gate events (zero or more), otherwise $g(id)$ denotes the gate events and begin/end events for all instances involved in id. I.e. since a ref interaction fragment is only a reference to another diagram name, it does not have internal events belongs to object instances. For example, consider SD_M shown in

Figure 3.10. For the ref interaction fragment $y \in F$ where $f(y) = (ref, 1)$, and for the object instance $b \in I$ there are no events associated with the operand and $g(y,1)_b = \emptyset$. However, for the environment instance $v_1 \in Env$, $g(y,1)_{Env} = \{e_6\}$ and $g(y) = \{e_6, e_7, e_8\}$.

For an interaction fragment $x \in F$ with an operand n and instance i, we write $min_i(g(x,n))$ to denote the minimal (first) event inside operand n for instance i. Similarly, $max_i(g(x,n))$ to denote the maximal (last) event inside operand n for instance i. When we omit the instance we refer to the subset of minimal/maximal events respectively. Consider the interaction fragment x shown in Figure 3.9 with the associated instances $a, b, c \in I$. Here, $g(x,1) = \{e_6, e_7\}$, $g(x,2) = \{e_8, e_9\}$ and $g(x) = \{e_3, \ldots, e_{12}\}$. The events associated with the beginning and end of the fragment for instance $a \in I$ is represented by $\overline{g(x)_a} = \{e_3, e_{10}\}$. Also, $min_a(g(x,1)) = e_6$ and $min_a(g(x,2)) = e_8$. Since each operand contains only one local transition, in this case $min_a(g(x,1)) = max_a(g(x,1))$ and $min_a(g(x,2)) = max_a(g(x,2))$.

Moreover, function $h(id, n)$ can be used to determine the set of nested interaction fragments inside the n^{th} operand of the interaction fragment id. Given functions f, g and h, we establish a few properties indicating that (1) a fragment can never be nested in itself (2) two arbitrary (not nested) interaction fragments do not share events, and (3) the events of an interaction fragment id_2 nested inside the n^{th} operand of another interaction fragment id_1 are contained in the set of events of that operand given by $g(id_1, n)$.

Function $j(id, n)$ is used to denote the subset of (object and environment) instances involved in fragment id's n^{th} operand. In a more general sense, and similarly to g, j is defined over pairs (id, n) where id is an interaction identifier and n is a natural number indicating the operand number. We use $j(d)$ to denote the set of instances associated with the whole diagram d where $j(d) \cap$

Env denotes the set of environment instances involved in gates at d level only. For instance in SD_G (Figure 3.9), $j(G) = \{a, b, c, v_1\}$, where $I_G = \{a, b, c\}$ and $Env_G = \{v_1\}$. For the interaction fragment $x \in F$, where $f(x) = (alt, 2)$: $j(x, 1) = \{a, b\}$ and $j(x, 2) = \{a, c\}$.

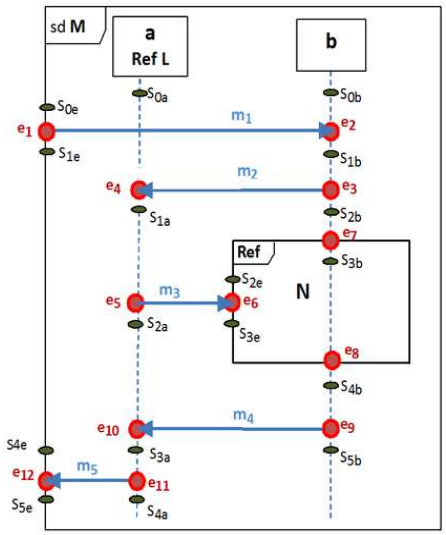
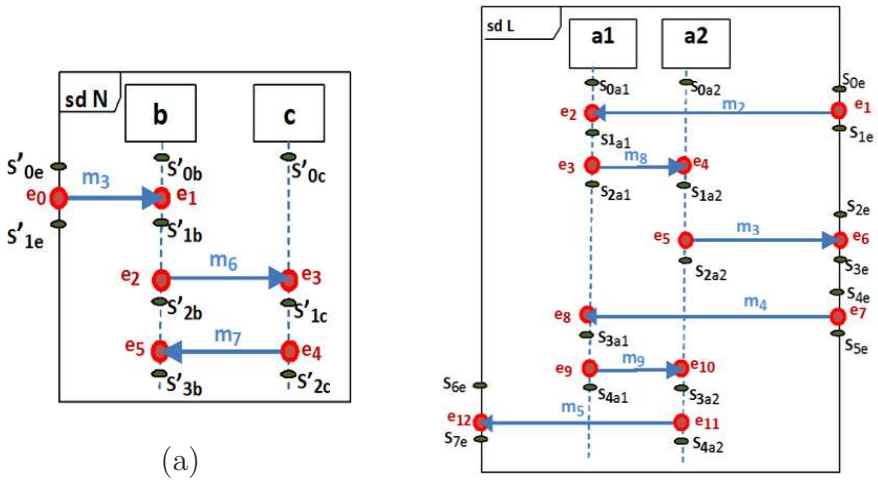


Figure 3.10: Different decomposition mechanisms in a sequence diagram.

Further, function ref is used to capture the decomposition of the diagram. It is defined over object instances to indicate *lifeline decomposition* and over fragment identifiers to indicate *interaction uses*. If ref is defined for an instance $i \in I_d$ with $ref(i) = n$ then the events in its lifeline denoted by E_i also belong to the set of events of the sequence diagram SD_n, that is, $E_i \subseteq E_n$. The instance i will be decomposed further in n and is therefore not an object instance of diagram n. In particular, all object instances involved in the reference fragment are instances of that diagram. Formally this means, if $ref(id) = n$ then $j(id) \cap I_d \subseteq I_n$ with $id \in F$, $n \in \mathcal{N}$.

Consider the examples modelled in Figure 3.10 and Figure 3.11, showing a SD named M and the two referred diagrams SD_N and SD_L. Diagram SD_M contains both forms of decomposition mechanisms available in UML2 sequence

diagrams, namely lifeline decomposition (instance a decomposed in sequence diagram L) and interaction use (reference to diagram N). The interaction involving instances a and b starts with message m_1 being sent (by the environment) and received by instance b. This triggers message m_2 being sent from b to a, followed by an interaction use to diagram N (with some input given by message m_3), and so on. The details of diagram N are described in the separate sequence diagram SD_N. To clarify the details of our formal model, we indicate all events and state locations explicitly along instance lifelines and frame lines (for gates). The diagram SD_M has three gates: the sending of m_1, the receiving of m_3 and the receiving of m_5.



(a)

(b)

Figure 3.11: The referred sequence diagrams by SD_M.

Formally, let $Env \supseteq \{v_1, v_2, v_3\}$ be the three environment instances involved in the gates of diagram M. SD_M is such that $I_M = \{a, b\}$, (E_M, $<_M$, M_M and T_M as shown in the figure), $F_M = \{id\}$ and ref is such that $ref(id) = N$ and $ref(a) = L$ (and otherwise undefined). Also $f(id) = (ref, 1)$, $g(id, 1) = \{e_6\}$, $g(id) = \{e_6, e_7, e_8\}$, $j(M) = \{v_1, v_3, a, b\}$, $j(id) = \{b, v_2\}$ and $j(id) \cap I_M = \{b\} \subseteq I_N$ as expected.

For the diagram SD_N in Figure 3.11 (a), $I_N = (\{b, c\}$, E_N, $<_N$, M_N and T_N, $F_N = \emptyset)$, Here, ref not defined for any $i \in I_N$, and $j(N) = \{v^{'}, b, c\}$ where $v^{'}$ is the environment instance is involved in the interaction. Similarly, the elements of SD_L can be defined.

Moreover, a UML 2 SD may contain variables and a given variable may be used several times in the same diagram. Finally, local transitions may have conditional statements associated with them which are given by the function *guard*. For example, when modelling alternative behaviour using an *alt* interaction fragment, each operand may be given a different condition or guard, which can be seen as associated with the first local transition in the operand. A *loop* interaction fragment also has an iteration condition associated with it. For example, in Figure 3.9, x is a local variable associate with the *alt* interaction fragment. In our formal representation, we associate the guard conditions with the first local transition in each operand of the interaction fragment such that $guard(t_1) = [x == 1]$ and $guard(t_2) = [x == 2]$, where $t_1 = (e_6, m_1, e_7)$ and $t_2 = (e_8, m_2, e_9)$.

Additionally, we introduce a set of *state locations* belonging to instances of a SD in order to represent the state of the instance before and after each event occurrence. A state location describes a situation during the life of an object after satisfying some activity or waiting for an event. When the state location belongs to an object instance, they are placed along a lifeline and in the case of an environment instance, state locations are places along the frame of a diagram or an interaction fragment. In our formal representation we define initial, internal and end state locations for each instance (see description below). Similarly to events, state locations cannot be shared by different instances, and are fully determined by functions μ, λ and θ.

The definition of state location considers the complete set of instances I^+

that includes both object and environment instances.

Definition 3.2 (State Location) *Let SD_d be a sequence diagram for a named diagram d. Its associated set of state locations is given by the set S_d:*

- $S_d = \bigcup_{i \in I^+} S^i$ *where* S^i *are the state locations for instance* i*, and* $S^i \cap S^j = \emptyset$ *for arbitrary* $i \neq j \in I^+$*;*

- $S^i = S^i_{ini} \cup S^i_{int} \cup S^i_{end}$ *is a set of initial, internal and end state locations for instance* $i \in I^+$ *respectively. Each instance has exactly one initial and one end state location. For* $i \in Env$*,* $S^i_{int} = \emptyset$*;*

- $\mu_i : M \times E_i \rightarrow S^i$ *is an* I^+*-indexed function that given a pair* (m, e) *of a message* m *and an event* e *associates it with a next state location of instance* i*. It is only defined if there is a* $t \in T$ *with* $t = (e, m, e^{'})$ *or* $t = (e^{'}, m, e)$*;*

- $\lambda_i : F \times \mathbb{N} \rightarrow 2^{S^i}$ *is an* I^+*-indexed function that given a pair* (id, n) *associates operand* n *of fragment* id *with a set of state locations of instance* $i \in j(id)$*, indicating all its state locations in the operand;*

- $\theta_i : F \rightarrow S^i$ *is an* I^+*-indexed function that given an interaction fragment* id *returns one state location for instance* $i \in j(id)$ *which is associated with the end of the fragment.*

For a given (object or environment) instance state locations and events are interleaved, whereby in the case of environment instances (described in the examples below) for each instance we only need in effect two state locations and one event. When the state locations belong to object instances the state locations and events are interleaved along a lifeline, whereby we always start and end with a state location. For each instance there is always a unique

start and end state location given by S^i_{ini} and S^i_{end} respectively, and object instances can furthermore have zero or more internal state locations given by S^i_{int}. The internal and end state locations are a result of local transitions or entering/leaving interaction fragments. On the other hand, the environment instances do not have internal state locations. The S^i_{ini} and S^i_{end} state locations belong to environment instances are places on the frame of the diagram or the interaction fragment depend on the situation they are used.

Consider SD_N shown in Figure 3.11, where events and state locations are indicated explicitly. The initial and end state locations for $v \in Env$ are $S^{'}_{0e}$ and $S^{'}_{1e}$, respectively. For $b \in I$, $S^{'}_{0b}$ is the initial state location and the set $\{S^{'}_{1b}, S^{'}_{2b}, S^{'}_{3b}\}$ contains internal state locations.

The effect of a local transition for an instance i is described by μ_i. That is, μ_i associates a unique state location (or an end state location, if the transition is the last interaction between two instances or i is an environment instance) to each message and event pair if this pair belongs to an existing local transition. That is when the μ_i for an environment instance i associated with a gate (necessarily involved in a local transition) returns the associated end state location for the instance. Thus, for each transition $t \in T$ with $t = (e_1, m, e_2)$ and where $e_1 \in E_i$ and $e_2 \in E_j$ we obtain two state locations $s_1 \in S^i, s_2 \in S^j$ associated with the two events in such a way that, $\mu_i(m, e_1) = s_1$ and $\mu_j(m, e_2) = s_2$. For self-transitions we obtain two state locations for each of the events and these belong to the same instance. For example consider the local transition $t = (e_2, m_6, e_3)$, $t \in T$ in Figure 3.11. The associated state locations $S^{'}_{2b}, S^{'}_{1c} \in S$ are such that $\mu(m_6, e_2) = S^{'}_{2b}$ and $\mu(m_6, e_3) = S^{'}_{1c}$.

The function λ_i, defined over pairs (id, n) of an interaction fragment identifier id and operand n, returns a set containing state locations for i. These state locations correspond to the state locations inside the given operand. It is

such that for each instance involved in an interaction fragment, each operand starts and finishes with a state location. In the case of reference fragments (only one operand) each instance has in fact a unique state location within that fragment. This applies to both object and environment instances. In other words, for any instance $i \in j(id)$ such that $f(id) = (ref, 1)$, $\lambda_i(x, 1)$ is a singleton indicating the unique state location for i in id. If $i \in j(id) \cap Env$, that is, i is an environment instance involved in fragment id, then there must be a gate event at id and λ_i returns one state location associated with that event (function θ_i returns the other one).

We use $min(\lambda_i(id, n))$ (and similarly $max(\lambda_i(id, n))$) to denote the first (last) state location in operand n of id. Overall, an interaction fragment has events marking the beginning and end of the fragment, for example $e_0, e_0^{'} \in \overline{g_i(id)}$ for instance i. Function θ is used to determine for each instance the state location that follows an interaction fragment. That is the function θ_i associates the next state location with the end of a fragment.

Recall SD_G shown in Figure 3.9 with four instances such that $j(G) = \{a, b, c, v_1\}$, where $I_g = \{a, b, c\}$ and $Env_g = \{v_1\}$. The interaction starts with message $m0$ being sent by the environment v_1 and received by instance a in such a way that $\mu(m0, e1) = S_{1e}$ and $\mu(m0, e2) = S_{1a}$. This follows by an interaction fragment $F = \{id\}$, with $f(id) = (alt, 2)$ and $j(id) = \{a, b, c\}$. Here, $S^a_{ini} = \{S_{0a}\}$, $S^a_{end} = \{S_{6a}\}$, $\lambda_a(id, 1) = \{S_{2a}, S_{3a}\}$, $\lambda_a(id, 2) = \{S_{4a}, S_{5a}\}$, and $\theta_a(id) = \{S_{6a}\}$. All non-initial state locations are determined by μ_i with $i \in I^+_G$.

Further, for interaction fragments with multiple operands, the state locations for a given instance inside an operand is defined, only if there are interactions involved in that instance. Here, we use the function $j(id, n)$ to identify the instances for a given operand. For example, in Figure 3.9, $j(id, 1) = \{a, b\}$ and

$j(id, 2) = \{a, c\}$. Therefore, there are no state locations within the operand 2 that belongs to instance b and no state locations within the operand 1 that belongs to c.

Further, recall SD_M shown in Figure 3.10. For the state locations we have $S^b_{ini} = \{S_{0b}\}$, $S^b_{end} = \{S_{5b}\}$, $S^{v_2}_{ini} = \{S_{2e}\}$ and $S^{v_2}_{end} = \{S_{3e}\}$, $\mu_b(m_1, e_2) = S_{1b}$, $\mu_{v_1}(m_1, e_1) = S_{1e}$, $\lambda_b(id, 1) = \{S_{3b}\}$, $\lambda_{v_2}(id, 1) = \{S_{3e}\}$, $\theta_b(id) = S_{4b}$ and $\theta_{v_2}(id) = \{S_{2e}\}$.

3.1.6 Regions in a Sequence Diagram

The decomposition mechanisms in a SD are particularly useful when modelling component-based systems. It provides a better structure to larger and complex interactions and consequently supports partial analysis, model evolution and incremental development.

Interactions in a SD can decompose and hide a set of interactions from a diagram with a high-level view. For example, SD_M in Figure 3.10 a detail representation of the interactions associate with instance a (lifeline decomposition instance) and ref interaction fragment are represented by SD_L and SD_N in Figure 3.11, respectively. Here, even the internal behaviour of the referred diagrams is fairly different, the behaviour in the context of the interaction in SD_M remains unchanged.

Consider SD_P shown in Figure 3.12. We may want to analyse a property of the diagram concerning only the interaction behaviour of instance $a1$ and $a2$ or a subset of interactions. For that, we introduce the notion of a *region* that facilitates the partial analysis of this interaction. Here, SD_P contains two regions, the set of interaction concerning the communication between b and c (region 1) and the interaction isolating instances a_1 and a_2 (region 2). In order to illustrate this notion we show it explicitly using a dashed-line around

the considered sub-interaction. The region contains not only events involved in the sub interaction but the underlying instances.

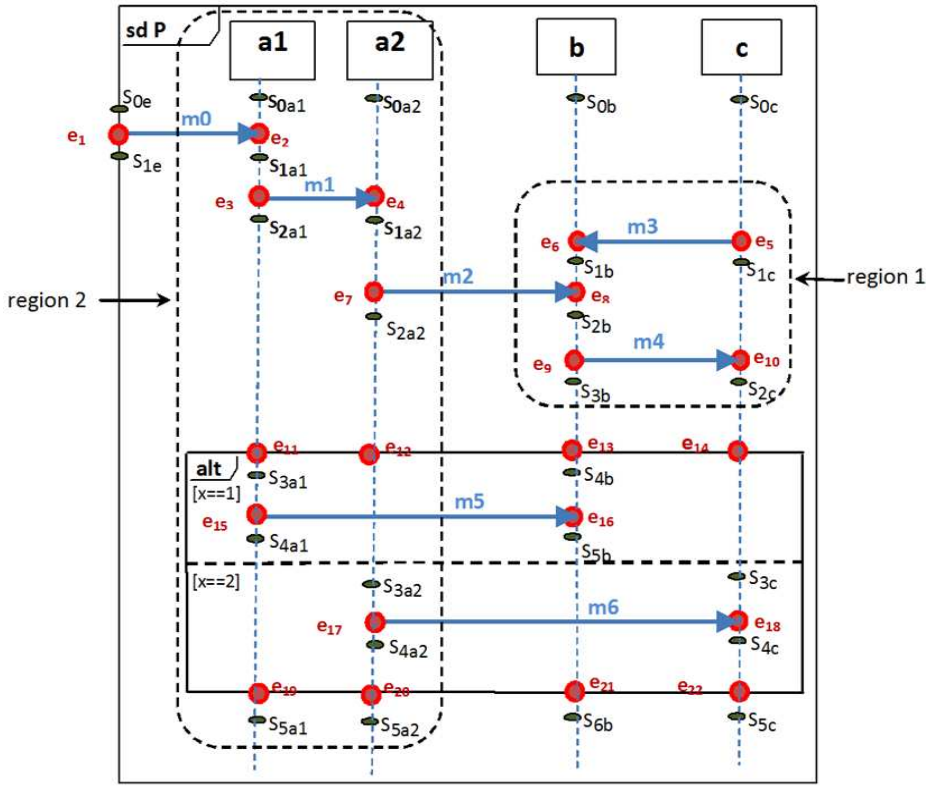


Figure 3.12: A sequence diagram with regions.

In particular, these regions can be considered as separate SDs and can thus be transformed into a CPN for analysis separately. For example, in Figure 3.12, let the interaction within the region 1 and 2 can be captured by separate sequence diagrams SD_R (Figure 3.13) and SD_T (Figure 3.14), respectively.

SD_R in Figure 3.13 consists of instances $b, c, \in I$ and their interactions represented by the local transitions $t_1, t_2, t_3 \in T$. Here, the source event of t_2 is considered as a gate.

SD_T in Figure 3.14 shows interactions associate with instances $a1$ and $a2$ only. In this example, gates are used to denote the source or target of the

interactions, where the associated instance is unspecified within SD_T. These events belong to environment instances such that $e_1, e_6, e_{10}, e_{12} \in E_{Env}$.

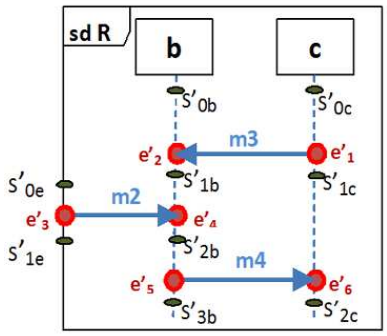


Figure 3.13: The region 1 of SD_P (Figure 3.12) as a sequence diagram.

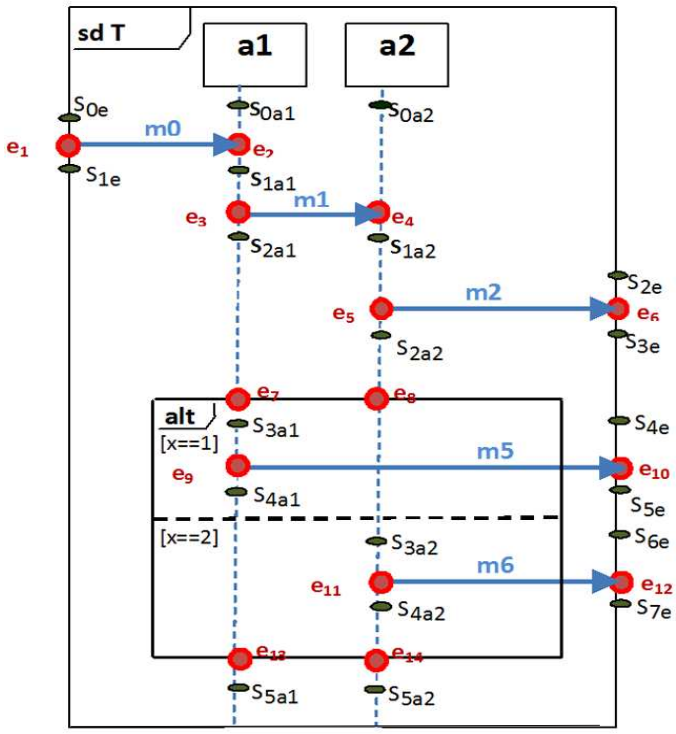


Figure 3.14: The region 2 of SD_P (Figure 3.12) as a sequence diagram.

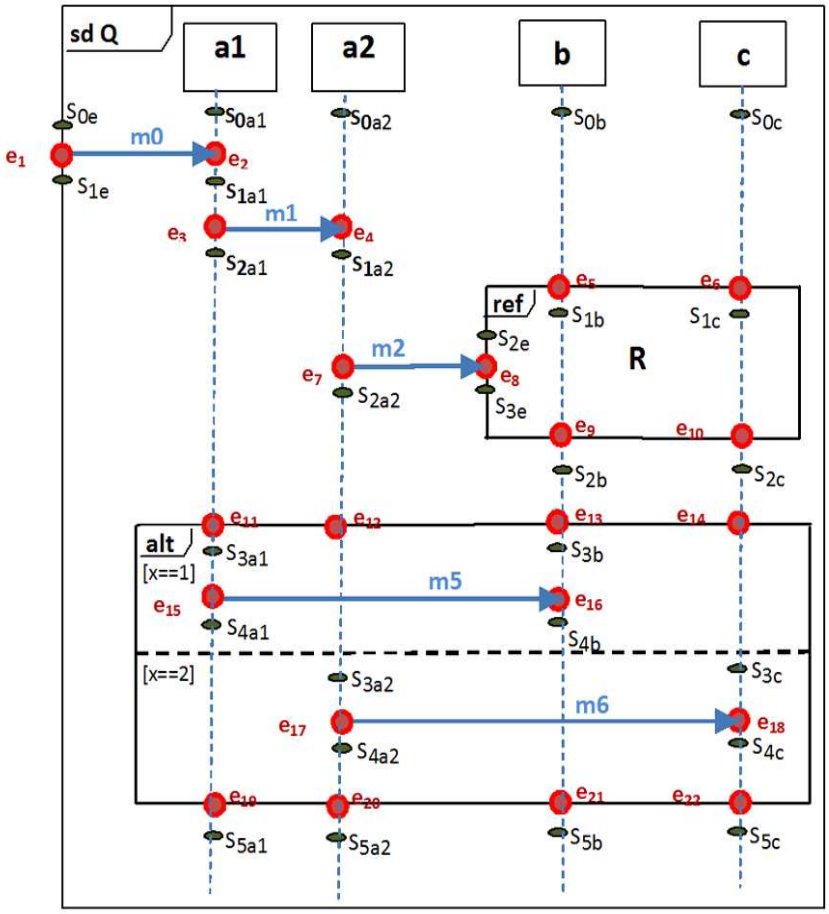


Figure 3.15: Replacing region 1 of SD_P (Figure 3.12) by an interaction use.

Moreover, considering regions as SDs can change the representation of the original SD. I.e. each identified region can be replaced by a ref interaction fragment or ref function as appropriate. Here, if a region contains the entire lifeline events of one or more instances, then we use lifeline decomposition. Otherwise, we consider interaction use. For example, by representing the region 1 of SD_P (Figure 3.12) using SD_R (Figure 3.13) we can replace the corresponding interaction of the original diagram by an interaction use as shown in Figure 3.15. Similarly, SD_S in Figure 3.16 shows the replacement of region 1 and 2 by interaction use and lifeline decomposition, respectively.

In order to describe the relationship among these diagrams, consider SD_Q shown in Figure 3.15. The interaction fragment $x \in F$: $f(x) = (ref, 1)$ refers the diagram SD_R in Figure 3.13 such that $ref(x) = R$. The interaction in SD_R represents the sub-interaction given by region 1 in SD_P shown in Figure 3.12. In SD_Q the local transition $t = (e_7, m_2, e_8)$ connects to ref using a gate.

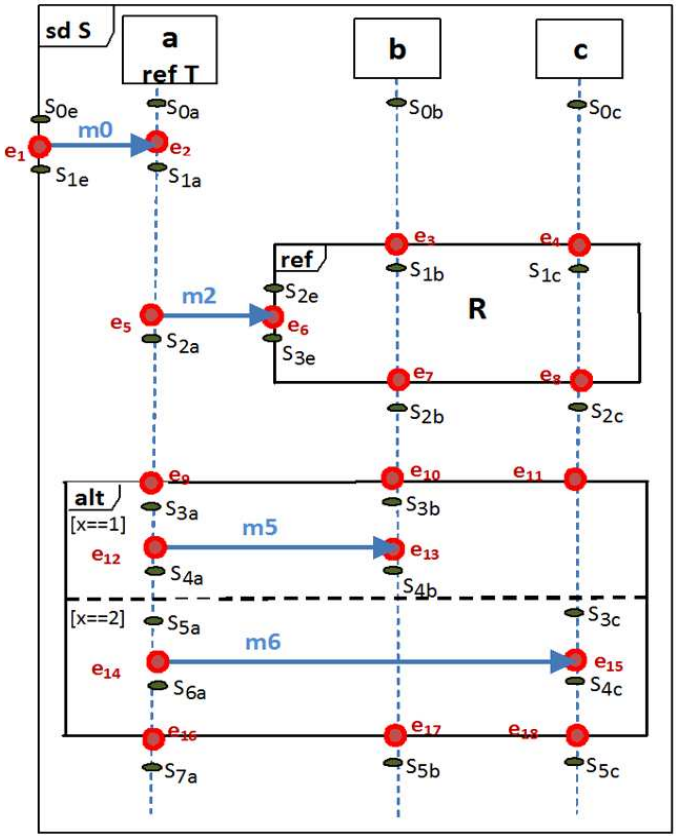


Figure 3.16: Replacing region 1 and 2 of SD_P (Figure 3.12)

Similarly, consider SD_S shown in Figure 3.16 that replace the region 1 and 2 in SD_P shown in Figure 3.12 using *interaction use* and *lifeline decomposition*, respectively. In SD_S, the interactions associate with the instance $a \in I$ refers the interaction in SD_T (Figure 3.14) such that $ref(a) = T$. The ref interaction fragment $x \in F$, refers to the interactions in SD_R (Figure 3.13) such that

$ref(x) = R$. Consequently, by replacing the referred interactions of SD_S by the diagrams SD_R and SD_T, we can obtain an interaction behaviour similar to the original diagram SD_P.

In order to extend our approach to deal with partial and modular synthesis, we formally define a region over a SD as itself a SD and defined as follows.

Definition 3.3 (Region) *Let SD_d be a sequence diagram. A region over SD_d is a triple $R = (I_r, T_r, F_r)$ of instances, local transitions and interaction fragment identifiers, such that $I_r \subseteq I_d$, $T_r \subseteq T_d$ and $F_r \subseteq F_d$.*

A region over a sequence diagram d as defined above consists of an arbitrary subset of instances, local transitions and interaction fragments. Consider SD_P shown in Figure 3.12. The region 1 can be formally represented as $R_R = (I_R, T_R, F_R)$ where $I_R = \{b, c\}$, $T_R = \{t_1, t_2, t_3\}$ and $F_R = \emptyset$. Similarly, region 2 can be represented as $R_T(I_T, T_T, F_T)$ where $I_T = \{a1, a2\}$, $T_T = \{t_1, \cdots, t_5\}$ and $F_T = \{x\} : f(x) = (alt, 2)$.

In order to define a more specific notion, we are interested in regions that are designate *fragment* and *order-closed*. A *fragment-closed* region means that a fragment is always completely enclosed in the region. On the other hand, the *order-closed* regions are such that if two events of a certain instance belong to a region, so do all intermediate events. We define a fragment-closed and order-closed region separately as follows.

Definition 3.4 (Fragment-closed Region) *Let SD_d be a sequence diagram, and $R = (I_r, T_r, F_r)$ a region over SD_d. R is fragment-closed iff for any $id \in F_r$ and $t = (e_1, m, e_2) \in T_d$ with $e_1, e_2 \in g(id, n)$ for some $n \in \mathbb{N}$ then $t \in T_r$.*

For the next definition, assume that a local transition t *contains* an event e iff $t = (e, m, e_1)$ or $t = (e_1, m, e)$. A region $R = (I_r, T_r, F_r)$ has an associated set of *region* events E_r and *region gate* events E_{G_r} defined as follows: for

any $e_1, e_2 \in E_r$ with $e_1, e_2 \in E_{d_i}$ and $i \in I_r$. For all $e_3 \in E_{d_i}$ such that $e_1 <_d e_3 <_d e_2$ necessarily $e_3 \in E_r$. When the corresponding instance of a source or target event of a local transition does not contain in the region, then the event becomes a gate.

Definition 3.5 (Region Event) *Let SD_d be a sequence diagram, and $R = (I_r, T_r, F_r)$ be a region over SD_d. An event $e \in E_d$ is a* region event *for R iff there is a $t \in T_r$ containing e and $e \in E_{d_i}$ with $i \in I_r$ or $e \in g\overline{(id, n)}$ for some $id \in F_r$ and $i \in I_r$. If e is contained in a local transition $t \in T_r$ but $e \in E_{d_i} : i \notin I_r$ we call the event a* region-gate event. *The set of region events (region-gate events) associated with R is given by E_r (E_{G_r}).*

Definition 3.6 (Order-closed Region) *Let SD_d be a sequence diagram, and $R = (I_r, T_r, F_r)$ be a region over SD_d. R is order-closed iff for any $t \in T_r$ with $t = (e_1, m, e_2)$ where $e_1 \in E_{d_i}$, $e_2 \in E_{d_j}$ for some $i, j \in I_d$ then $i, j \in I_r$ and $e_1 <_r e_2 \in E_{r_i}$, if $e_1 <_d e <_d e_2$ then $e \in E_{r_i}$.*

Consider SD_P given in Figure 3.17. The regions indicated correspond to $R_1 = (\{b, c\}, \{t_2, t_3, t_4\}, \emptyset)$, and $R_2 = (\{a_1, a_2\}, \{t_0, t_1, t_2, t_5, t_6\}, \{id\}) : t_k = (e_i, m_k, e_j) \in T_r)$.

$R_1 = (\{b, c\}, \{t_2, t_3, t_4\}$, where $t_2 = (e_7, m_2, e_8)$, $t_3 = (e_5, m_3, e_6)$, and $t_4 = (e_9, m_4, e_{10})$ is both *fragment-closed* and *order-closed*. R_1 is trivially *fragment-closed* as it does not contain an interaction fragment such that $F_{r_1} = \emptyset$. R_1 is *order-closed* because $E_{r_1} = \{e_5, e_6, e_8, e_9, e_{10}\}$ and there is no event e in E_{P_b} or E_{P_c} in between any of the events of E_{r_1}.

Similarly, $R_2 = (\{a_1, a_2\}, \{t_0, t_1, t_2, t_5, t_6\}, \{id\})$ is fragment closed by definition as the region contains all the local transitions within the fragment *alt*. Also R_2 is *order-closed* because $E_{r_2} = \{e_1, e_2, e_3, e_4, e_7, e_{11}, e_{12}, e_{15}, e_{17}, e_{19}, e_{20}\}$. R_2 is closed for causality in a_1 and a_2. Here, $E_{G_{r_2}} = \{e_8, e_{16}, e_{18}\}$ are region-gate events.

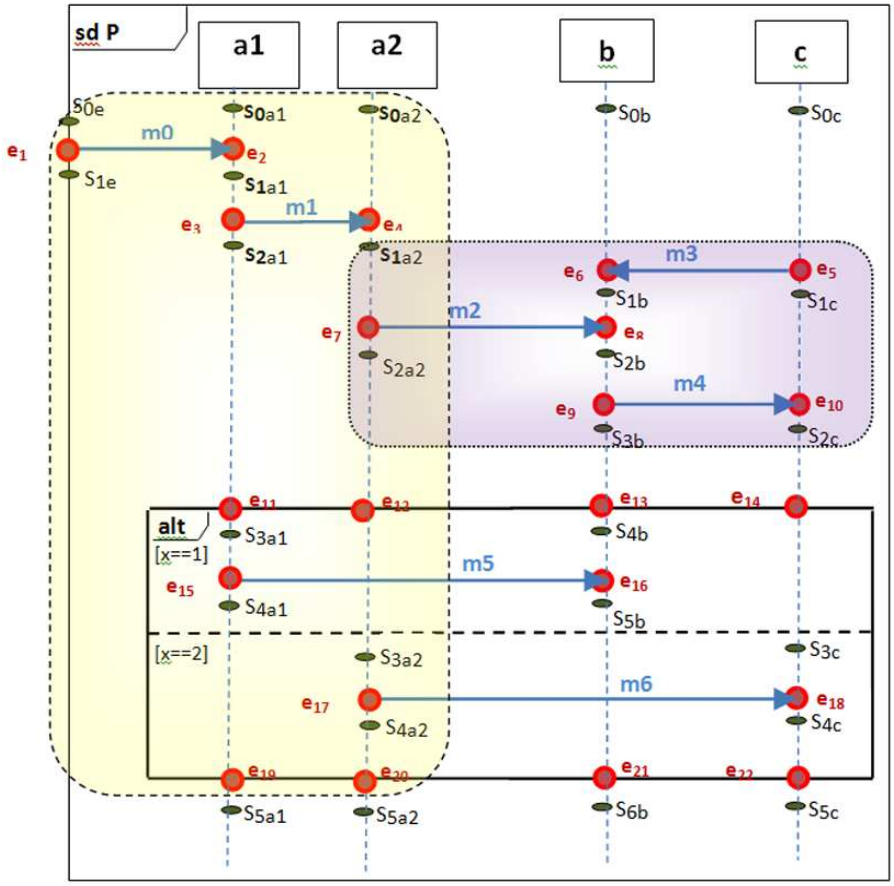


Figure 3.17: A sequence diagram with fragment-order closed regions.

Fragment and order-closed regions are called *closed* regions for short. A closed region over a SD d is itself a SD contained in d with sets and relations. For example, consider the sequence diagrams SD_P and SD_R shown in Figure 3.12 and Figure 3.13, respectively. The region $R_1 = (\{b, c\}, \{t_2, t_3, t_4\}, \emptyset)$, in SD_P determines the sequence diagram SD_R. Consider the event $e_7 \in E_{P_{a2}}$ where $a_2 \notin I_r$. Here, we consider e_7 as a gate event for the local transition $t_2 = (e_7, m_2, e_8)$ in SD_P. This reflects in SD_R where the corresponding event e_3' belongs to an environment instance.

The notion of a *closed* region defines as follows.

Definition 3.7 (Closed Region SD) *A closed region* $R = (I_r, T_r, F_r)$ *over* SD_d *determines a sequence diagram* $SD_r = (I_r, E_r, <_r, M_r, T_r, F_r, ref_r)$ *where* $E_r = \{e_1, e_2 \mid t = (e_1, m, e_2) \in T_r\} \cup \bigcup_{f \in F_r, i \in I_r} g_i(f)$ *satisfying if* $e \in E_r \cap E_{d_i}$ *and* $i \notin I_r$ *then* $e \in E_{r_s}$ *for some* $s \in Env$, $<_r \subseteq <_d$, $M_r = \{m \mid t = (e_1, m, e_2)$ for some $t \in T_r\}$, *and* $ref_r \subseteq ref_d$.

Further, consider a sequence diagram SD_R (see Figure 3.13) that is determined by a closed region R over a sequence diagram SD_P (see Figures 3.12). Here, SD_R is the reference in a sequence diagram SD_S (see Figure 3.16), where SD_S is behaviourally equivalent to SD_P. Generally, the reference is either an interaction use or a lifeline decomposition. Here, we assume that if a region is such that it contains all the events for the instances in the region (i.e., for any $i \in I_r$, $E_{r_i} = E_{P_i}$) then it corresponds to a lifeline decomposition, otherwise it is an interaction use. For interaction use, we add a new interaction fragment identifier to $x \in F_S$ such that $f(x) = (ref, 1)$ and $ref(x) = R$. For lifeline decomposition, we add a new instance to $j \in I_S$ such that this instance has its (internal) behaviour decomposed in a referred diagram (see SD_T in Figure 3.14) given by $ref_S(j) = T$.

3.1.7 Additional Functions in a Sequence Diagram

An additional function useful for defining our transformation rules later on is *next* indexed over instances in I^+ and defined over events and state locations. This function returns the *next* state locations/events (generally a singleton) for a given event/state location respectively.

Consider SD_F with alternative behaviour shown in Figure 3.18. The *next* function is defined as follows: $next_a(S_{0a}) = \{e_1\}$, $next_a(e_1) = \{S_{1a}\}$, $next_a(e_3) = \{S_{2a}, S_{5a}\}$, $next_a(S_{4a}) = \{e_{11}\}$, etc. Here the next state location of the event e_3 can be S_{2a} or S_{5a} based on the operand guard that evaluates

to true. Also, for the end state locations, $next_a(S_{7a}) = \{\perp\}$. We sometimes write $next_i(e) = s$ instead of $next_i(e) = \{s\}$ for simplicity.

We define the *next* function as follows.

Definition 3.8 (Function: next) *Let SD be a sequence diagram with set of state locations S. We define $next_i$ as an I^+-indexed function defined over state locations and events such that $next_i : S^i \cup E_i \to 2^{S^i \cup E_i}$. Let $id \in F$ be an arbitrary interaction fragment in the diagram with $f(id) = (o, m)$, and j be a natural number ranging $1 \leq j \leq m$.*

$$
next_i(x) = \begin{cases}
\{min(E_i)\} & \Leftarrow x \in S^i_{ini} \\
\{\mu_i(m, x)\} & \Leftarrow x \in E_i \text{ and } \mu_i(m, x) \\
 & \text{defined for some } m \in M \\
\{s_1, \ldots, s_m\} & \Leftarrow x \in E_i, \\
 & x = min(\overline{g_i(id)}) \text{ and} \\
 & s_j = min(\lambda_i(id, j)) \\
\{\theta_i(id)\} & \Leftarrow x \in E_i, \\
 & x = max(\overline{g_i(id)}) \text{ and} \\
 & o \neq break \\
\{max(\overline{g_i(id)})\} & \Leftarrow x \in S^i_{int}, \\
 & x = max(\lambda_i(id, j)) \text{ and} \\
 & o \neq \{par, loop\} \\
\{e^{'}\} & \Leftarrow x \in S^i_{int}, \text{ not covered} \\
 & \text{by the cases above with} \\
 & \mu_i(m, e) = x \text{ for some} \\
 & m \in M, e \in E_i \text{ and} \\
 & e \to e^{'} \\
\perp & \Leftarrow x \in S^i_{end}
\end{cases}
$$

For behaviour with par, loop and break interaction fragments we define $next_i$ as follows:

$$next_i(x) = \begin{cases} \{e^{'}\} \cup \bigcup_{p \neq j} g(id, p) & \Leftarrow x \in S^i_{int}, x \in \lambda_i(id, j), \\ & o = par, \\ & x \neq min(\lambda_i(id, j)), \\ & \mu_i(m, e) = x \text{ for some} \\ & m \in M, e \in E_i \text{ and} \\ & e \rightarrow e^{'} \\ \{min(\overline{g_i(id)}) \cup max(\overline{g_i(id)})\} & \Leftarrow x \in S^i_{int}, \\ & x = max(\lambda_i(id, j)) \text{ and} \\ & o = loop \\ \{\theta_i(id^{'})\} & \Leftarrow x \in E_i, \\ & x = max(\overline{g(id)_i}), \\ & o = break, \\ & f(id^{'}) = (loop, 1) \text{ and} \\ & h(id^{'}, 1) = id \end{cases}$$

The definition states that the *next* state location for an *event* is generally given by a singleton containing the state location determined by μ applied to the event if defined. If μ is not defined then either e is a beginning or an end event for an interaction fragment. If it is the beginning of an interaction fragment with m operands then the *next* state locations are given by the set of all first state locations for each operand. If it is an end event then the *next* state location is determined by θ.

Conversely, the *next* event for a *state location* depends on where the state location is. If it is inside a *par* fragment (but not the first state location of one of its operands) then the set of next possible events is more complex and given by the union of all events in the other fragments plus the next possible event within the operand. For example, consider the SD with parallel behaviour shown in Figure 3.21. There are two possible next events for the state location S_{3a} such that $next_a(S_{3a}) = \{e_7, e_9\}$.

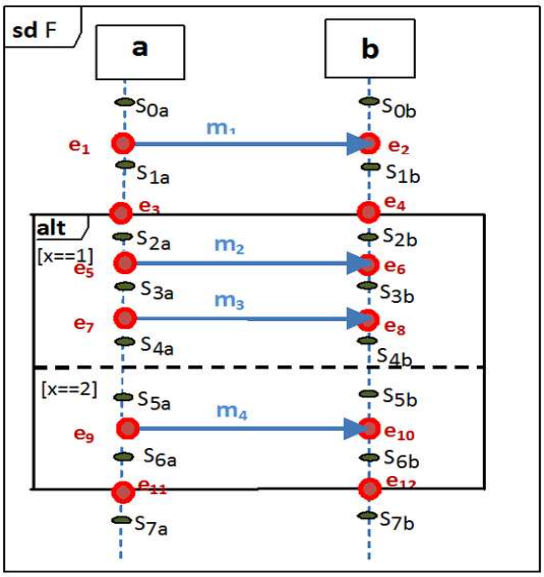


Figure 3.18: Illustrating the *next* function.

When the state location is the maximum of a *loop* fragment, there are two possible *next* events given by minimal or maximum event of the fragment based on the *guard* of the fragment. Also, when the fragment is a *break* nested within a *loop* fragment, the *next* state location of the maximum event of the *break* fragment is given by the θ of the *loop* fragment. For example, consider SD shown in Figure 3.19. Here, $next_a(S_{6a}) = \{e_1, e_{13})$ and $next_a(e_9) = S_{7a}$.

In all other cases, there is a unique next event given by: minimal event (for the initial state location), end event of the fragment (for the last state location in an operand of a non-*par* fragment), or the immediate following event (for all other cases). There is no next event for an end state location and this is applicable for state locations belong to both object and environment instances. For example, in Figure 3.21 $next_a(S_{2a}) = \{e_5\}$, $next_a(S_{5a}) = \{e_9\}$, $next_a(S_{4a}) = \{e_9, e_{11}\}$, $next_a(S_{7a}) = \{\perp\}$ and so on.

Recall SD_M shown in Figure 3.20 with reference behaviour. There is a unique *next* state location for a given event that belongs to an environment

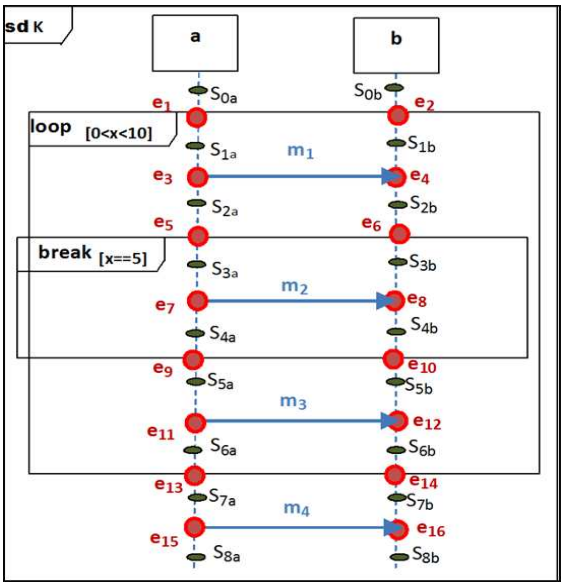


Figure 3.19: A sequence diagram with break behaviour.

instance. Such that $next_b(S_{0b}) = \{e_2\}$, $next_b(e_2) = \{S_{1b}\}$, $next_b(e_3) = \{S_{2b}\}$, $next_v(e_6) = \{S_{3e}\}$, $next_v(S_{3e}) = \{\perp\}$, so on for $v \in Env$.

Similarly, we can define a function $previous_i : S^i \cup E_i \rightarrow 2^{S_i \cup E_i}$, to extend the functions associated with the formal representation of a sequence diagram. However, we do not describe it in details here as it is not essential for this formal model. The following definition states the relationship between the *next* and *previous* function.

Definition 3.9 *Let $f(id) = (o, m)$, $e_1, e_2 \in g_i(id, n)$ be such that $e_1(e_2)$ is the minimal (maximal) event in the set of events associated with the operand $n \leq m$ of id for instance i, and $e_0(e_0^{'})$ be the beginning(end) event of id for instance i, that is $e_0, e_0^{'} \in \overline{g_i(id)}$. We can define the relationship between the next and previous function as follows:*

$$\lambda_i(id, n) = \begin{cases} \{s_1, s_2\} & \Leftarrow next_i(s_1) = e_1, previous_i(s_2) = e_2, \\ & \quad previous_i(s_1) = e_0, next_i(s_2) = e_0^{'}, n \leq m \\ \perp & \Leftarrow otherwise \end{cases}$$

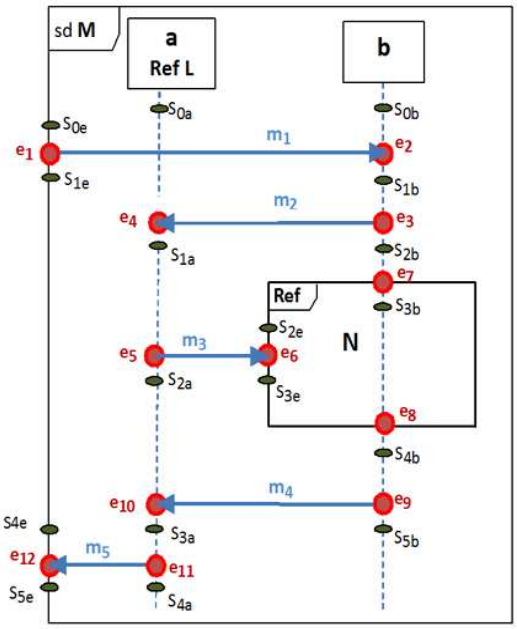


Figure 3.20: A sequence diagram with reference behaviour.

Consider SD_M shown in Figure 3.20. The relation between *next* and *previous* can be explained as follows: $next_b(S_{0b}) = \{e_2\}$, $next_b(e_2) = \{S_{1b}\}$. $previous_b(S_{1b}) = e_2$ and $previous_b(e_2) = \{S_{0b}\}$.

3.1.8 Trace in a Sequence Diagram

The notion of chains of interleaved state locations and events can be obtained for each object involved in the interaction and derived from the function *next* as expected. Interleaving means the merging of two or more traces such that the events from different traces may come in any order in the resulting trace, while events within the same trace retain their order. (Trace is defined later on). For example, consider SD_F in Figure 3.18 with *alt* behaviour, the instance a contains two chains $S_{0a} \cdot e_1 \cdot S_{1a} \cdot e_3 \cdot S_{2a} \cdot e_5 \cdot S_{3a} \cdot e_7 \cdot S_{4a} \cdot e_{11} \cdot S_{7a}$ and $S_{0a} \cdot e_1 \cdot S_{1a} \cdot e_3 \cdot S_{5a} \cdot e_9 \cdot S_{6a} \cdot e_{11} \cdot S_{7a}$.

Instead of an *alt* fragment, the parallel behaviour in Figure 3.21 has differ-

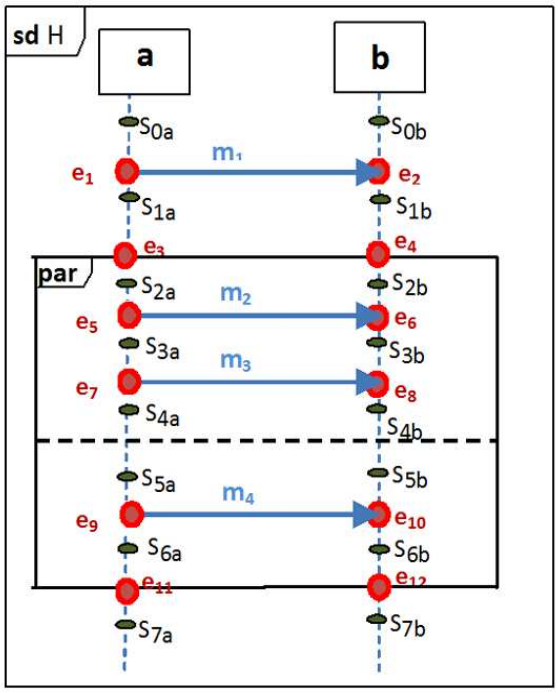


Figure 3.21: Illustrating chains.

ences in the *next* definitions. For example $next_a(S_{4a}) = \{e_9, e_{11}\}$) and we can derive three chains for instance a given by $S_{0a} \cdot e_1 \cdot S_{1a} \cdot e_3 \cdot S_{2a} \cdot e_5 \cdot S_{3a} \cdot e_7 \cdot S_{4a} \cdot e_9 \cdot S_{6a} \cdot e_{11} \cdot S_{7a}$, $S_{0a} \cdot e_1 \cdot S_{1a} \cdot e_3 \cdot S_{5a} \cdot e_9 \cdot S_{6a} \cdot e_5 \cdot S_{3a} \cdot e_7 \cdot S_{4a} \cdot e_{11} \cdot S_{7a}$, and $S_{0a} \cdot e_1 \cdot S_{1a} \cdot e_3 \cdot S_{2a} \cdot e_5 \cdot S_{3a} \cdot e_9 \cdot S_{6a} \cdot e_7 \cdot S_{4a} \cdot e_{11} \cdot S_{7a}$, These examples illustrate that the initial event of a fragment (here e_3) has several next state locations (S_{2a} and S_{5a}). However the state locations inside a *par* fragment have several possible next events. For example $next_a(S_{3a}) = \{e_7, e_9\}$.

In particular, the function *next* uses to define notions of SD *trace* and *language*. We return to such considerations later and use them to prove the correctness of model transformations in Chapter 7.

The idea of chains on state locations and events can be used to derive a notion of trace over message labels. The formal representation of the notion of a chain can be defined as follows.

Definition 3.10 (Chain) *Given a sequence diagram SD and associated set of state locations S, a chain c for $i \in I$ is a finite sequence of interleaved state locations and events of i such that $c = s_0 \cdot e \cdot \cdots \cdot s_j \cdot e^{'} \cdot s_k \cdot e^{''} \cdot \cdots \cdot s_f$ where $s_0 \in S^i_{ini}, s_f \in S^i_{end}, s_j, s_k \in S^i_{int}, e, e^{'}, e^{''} \in E_i, e = min(E_i)$, $e^{'} \in next_i(s_j), s_k \in next_i(e^{'})$ and $e^{''} \in next_i(s_k)$. Further, for an arbitrary event in a chain, say e_1, if $e_1 = max(\overline{g(id)}_i)$ with $f(id) = (par, m)$ then for all events $r \in g(id)$ with $r \neq e_1$, r must occur in the chain before e_1.*

From a chain a sequence of message labels over M can be obtained as follows: for every event $e \in E_i$ in a chain, if $\mu_i(m, e)$ is defined then take m and move to the next event, else move to the next event in the chain. Consequently, the chains in Figure 3.18 and Figure 3.21 correspond respectively to the following sequences of message labels: for the *alt* case $m_1 \cdot m_2 \cdot m_3$ and $m_1 \cdot m_4$, for the *par* case $m_1 \cdot m_2 \cdot m_3 \cdot m_4$, $m_1 \cdot m_4 \cdot m_2 \cdot m_3$, and $m_1 \cdot m_2 \cdot m_4 \cdot m_3$.

Generally, a trace is a sequence of events ordered by time that can be partial or total ordered [Micskei and Waeselynck, 2010]. A trace describes the information about a list of message exchanges corresponding to a system run. The trace-semantics describe the semantics of interactions.

We define the alphabet L_1 of a sequence diagram SD over the set of message labels M. The associated language L(SD), for a set of legal traces of the SD is defined as follows.

Definition 3.11 (Trace) *A trace of a sequence diagram SD with set of state locations S is a possibly infinite word w, $w = m_1 \cdot m_2 \cdot m_3 \ldots$ over the alphabet L_1 iff there exists a chain c of state locations and events for some instance $i \in I^+$. We can derive w from c by considering the message labels associate with the local transition that corresponds to an event such that $m = l(t)$ where $t = (e, m, e^{'})$ for $e \subset c$.*

Definition 3.12 (Language) *A language of SD is the set $\mathcal{L}_1(SD)$ of words over the alphabet L_1, where $\mathcal{L}_1(SD) = \{W |\ W$ is a maximal trace of $SD\}$. A trace is maximal if it is not a proper prefix of any other trace.*

The formal representation for a UML 2 SD described in this chapter is flexible to extend to other associated formal considerations. This may include different variants of a SD or different model transformation approaches such as incremental transformation, parametric transformation, and so on. (described in Chapter 6). The next sub-section 3.1.9 describes the formal representation for two variants associated with sequence diagrams.

3.1.9 Variants of a Sequence Diagram

For certain kinds of systems we may want to add (and verify) quantitative temporal constraints over an interaction. This section describes time and stochastic aspects associated with UML2 sequence diagrams and extends the formal semantics defined in Section 3.1.5 with time and stochastic constraints.

UML 2 standard [Arlow and Neustadt, 2005, Douglass, 2004, OMG, 2011a] describes time aspects associated with SDs using parameterised messages and assigning time stamp on event occurrences. Also, there are notion of a timers and a system clock that can produce interrupt events. Also, there is a textual language within UML called Object Constraint Language (OCL) that can be used to capture temporal constrains such as the specification of deadlines, durations, response times, delays, etc. if extended appropriately. Indeed, there are real-time extensions of OCL [OMG, 2006, Garousi, 2010, Lano, 2009].

The timing aspects associated with a SD can be used to indicate the start time, the time taken by an interaction, or the time interval between event occurrences on lifelines. In a SD, timing constraints bound the occurrence of (pairs of) events. In this thesis, we only allow two kinds of constraints:

between events from different lifelines if the events are associated with a local transition, or between the consecutive events on the same lifeline. For example consider SD_K in Figure 3.22. The constraint $\{0..2\}$ denotes the duration of a communication between two instances that bounds the occurrence of the corresponding send and receive events of the message ($t = (e_1, m_1, e_2)$). A further example is a timing constraint $\{1..3\}$ between events e_6, e_9 on the same lifeline which imposes a constraint on the behaviour of the corresponding instance.

Timing constraints are usually given by a number to indicate a *fixed delay* or time intervals (with upper and lower bounds) to indicate an *interval delay* [Ameedeen et al., 2011, OMG, 2011a]. Examples of possible notation include $\{n\}$ for a fixed delay of n time units, and $\{n_1..n_2\}$ for an interval delay between n_1 and n_2 time units, where $n, n_1, n_2 \in \mathbb{R}$. Further, the timing constraints can be specified using both integer and real numbered values. The formal representation of timing aspect of a SD can be introduced using a labelling function on events in the diagram. We define timing annotations as follows.

Definition 3.13 *Let SD_d be a sequence diagram. A timing function over SD_d is given by $time_{SD_d} : E \times E \to \mathbb{R}_0^+ \times \mathbb{R}_0^+$ and such that $time_{SD_d}(e_1, e_2)$ is only defined if $e_1 < e_2 \in E_i$ for some $i \in I+$ or there is a local transition $t \in T$ such that $t = (e_1, m, e_2)$. A set of timing annotations $\mathcal{T}$ over SD_d is given by $\mathcal{T} = \{\tau \mid \tau = (e_1, e_2, time_{SD_d}(e_1, e_2))$ with $e_1 < e_2 \in E_i$ or $\tau = (t, time_{SD_d}(e_1, e_2))$ if there is a $t = (e_1, m, e_2) \in T\}$.*

Figure 3.22 shows an example of a sequence diagram SD_K with two timing annotations, $\mathcal{T} = \{(t_1, [0, 2]), (e_6, e_9, [1, 3])\}$. That is the local transition $t_1 = (e_1, m_1, e_2)$ is associated with a timing constraint given by the function $time_{SD_k}(e_1, e_2) = [0, 2]$. Also, as specified by the diagram, the interval between

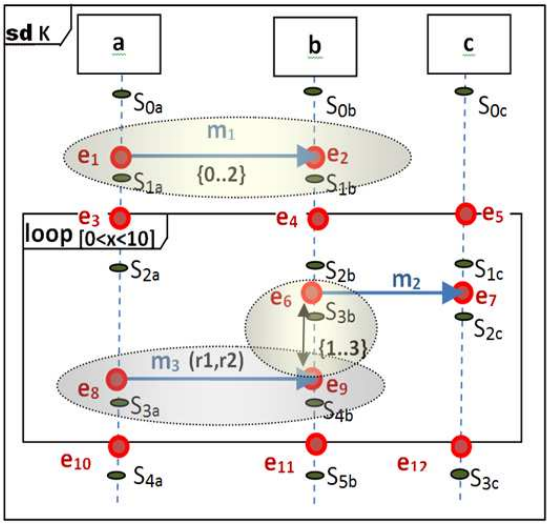


Figure 3.22: A sequence diagram with time and stochastic annotations.

the events e_6 and e_9 within the *loop* interaction fragment is specified using the function $time_{SD_k}(e_6, e_9) = [1, 3]$.

In addition, SDs can be modelled with stochastic delays specified by probability distributions. Such approaches are commonly used for performance evaluation [Bowles and Kloul, 2010, Emadi and Shams, 2009a, Garousi, 2010, Merseguer and Campos, 2004]. For most models this corresponds to having time in the transitions and the enabling time of such transitions specified by a distribution. The stochastic annotations for a SD considered in this thesis are given as rates over local transitions. The rate information corresponds to the movement of an object between two instances and can be used to capture performance aspects of a system.

A rate can be any positive real number (determining the negative exponential distribution) or the distinguished symbol $\top$ (indicate as unspecified). The rate is specified as $\mathbb{R}^{+}_{\top}$ for the set of positive real numbers together with the symbol $\top$. Local transitions denote communication between two instances and

each instance can constrain the delay of the communication. Local transitions can therefore be associated with a pair of rates $(r1, r2)$ where $r1$ corresponds to the rate associated with the sender and $r2$ to the receiver. If one of the rates is $\top$, the corresponding instance is passive, and the rate of the transition is uniquely determined by the other instance. If both rates are specified, it is usual to take the minimum of both as the transition rate that gives the synchronised rate associated with the interaction. This section introduces a labelling function on a local transition to indicate the rates associated with the sending and receiving instances. We define stochastic annotations as follows.

Definition 3.14 *Let SD_d be a sequence diagram. A rate function over SD_d is given by $rate_{SD_d} : T \to \mathbb{R}^+_\top \times \mathbb{R}^+_\top$. A set of stochastic annotations $\mathcal{S}$ over SD_d is given by $\mathcal{S} = \{\sigma \mid \sigma = (t, rate_{SD_d}(t))$ with $t = (e1, m, e2)\}$.*

Consider the example shown in Figure 3.22, where SD_K has stochastic annotations given by $\mathcal{S} = \{(t_3, (r1, r2))\}$. Here, the local transition $t_3 = (e_8, m_3, e_9)$ is associated with stochastic aspects, where the sending rate is $r1$ and the receiving rate is $r2$, $r1, r2 \in \mathbb{R}^+_\top$. This is given by the function $rate_{SD_k}(t_3) = [r1, r2]$.

As the formal model of the SD is extended with the time and stochastic aspects, new syntax and semantics can be added to the model for enhancing the expressiveness power of the model. The next section describes Interaction Overview Diagrams that facilitate to represent the hierarchical view of a SD as a possible extension to the discussed model.

3.2 Interaction Overview Diagram

Interaction Overview Diagrams (IODs) are introduced in UML 2 to improve the expressiveness and the structure of a given system design, by visualising the overall control flow of a system. An IOD provides a high-level structuring mechanism for the possible interactions in a system by combining sequence diagrams and activity diagrams (ADs) [Pilone and Pitman, 2005,OMG, 2011a]. IOD is a special and restricted kind of AD. Semantically, however, IODs and ADs are given different interpretations. IODs follow trace semantics similar to SDs. However, IODs are used to model the intra-object behaviour and SDs are used to model the inter-object behaviour. IODs are used to compose scenarios through sequence, iteration, concurrency or choice without showing all the detail of the lifelines and messages [Pilone and Pitman, 2005, OMG, 2011a]. Hence, IODs are used to reduce the complexity of a design model and represent a clear picture of the control flow of the system.

This section briefly describes the annotations and the formal representation of an IOD that need to define the hierarchical structure of a sequence diagrams described in Section 3.1. This section does not describe all the features of an IOD, as they are out of the scope of this thesis and will be left for future work.

3.2.1 Main Notations of an Interaction Overview Diagram

IOD uses activity diagram notations to define the control flow of the interactions, where the activity nodes are either inline interactions (SDs) or interaction uses (ref interaction fragments). IOD describes interactions in such a way that the messages and lifelines are abstracted away. Here, an IOD does not itself show the involved lifelines or messages even though the lifelines may occur explicitly within inline interactions in the activity nodes.

The high-level structure of an IOD composes scenarios through mechanisms

such as sequence, iteration, concurrency or choice. In order to trace these behaviours, IOD incorporates activity diagram notations such as fork, join, decision and merge nodes. However, branching and joining of branches in an IOD must be properly nested, which is more restrictive than in an activity diagram.

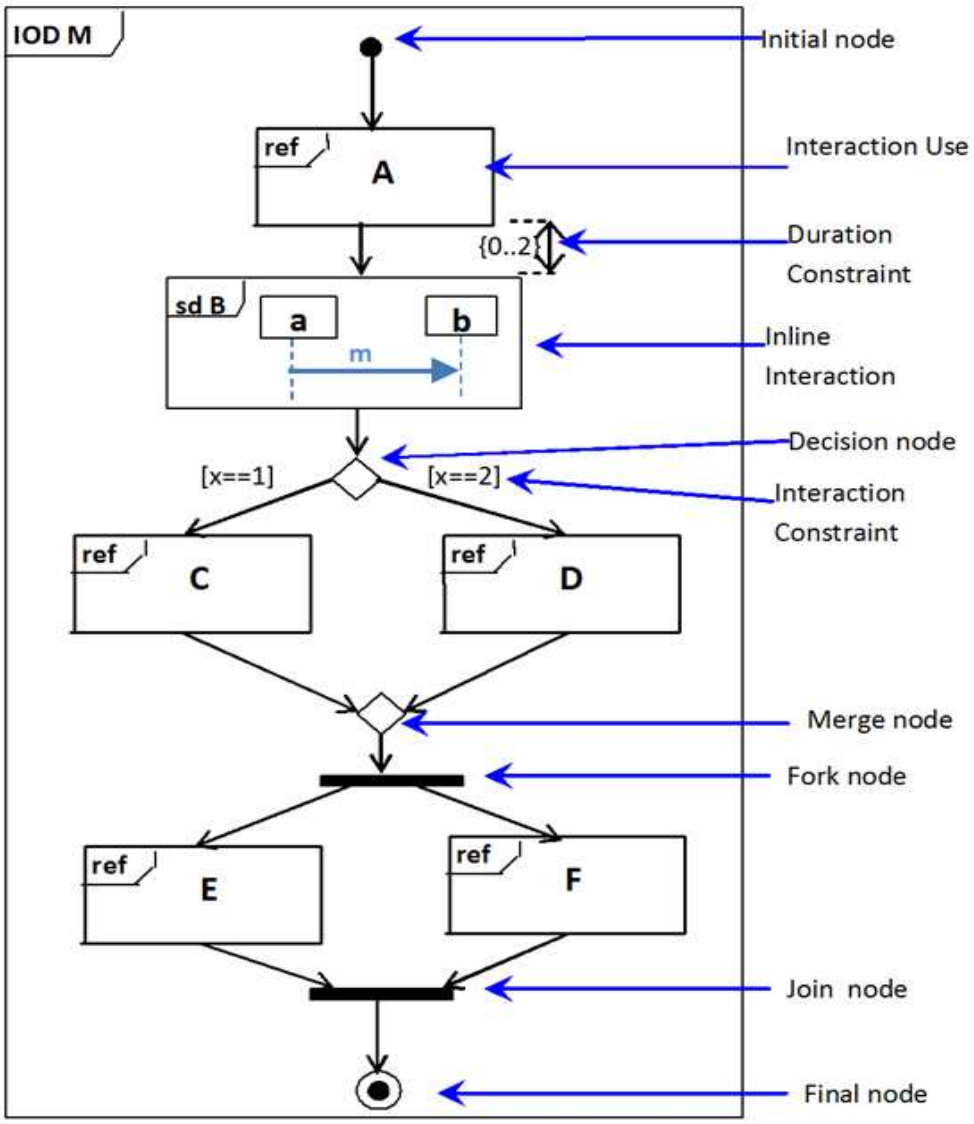


Figure 3.23: An example of an interaction overview diagram.

The main elements of an IOD are shown in Figure 3.23 and can be described as follows. An IOD is represented using a rectangular frame around the diagram with a name in a compartment in the upper left corner. There is an entry point and an exit point for an IOD named initial node and final node, respectively. Generally, an initial node contains tokens that facilitate the execution between the nodes. (We do not consider tokens in detail in this thesis). The initial state is shown as a small solid filled circle and the exit point is shown as a small filled circle within a large circle. An IOD may contain more than one final node, such that the activity flow stops when it reaches to the first final node.

The control flow within the diagram is shown using directed arrows between the nodes. A SD of any kind or an *interaction use* may appear inline as the activity nodes of an IOD for activity invocation. When the node is an inline interaction (see SD_B), the behaviour within the interaction will be executed. When the node is an *interaction use* (eg. A, C, D, E, F), the inline interaction will be replaced by the occurrence specified using the name of the referred interaction by a replica of the interaction.

An IOD contains a set of control nodes that supports the control flow of the model (described in Figure 3.23). A fork node is a control node that splits a flow into multiple concurrent flows. That is, it has a single incoming flow and two or more outgoing flows, where the incoming tokens are offered to all outgoing flows (edges). This indication of parallel behaviour is represented by a solid bar with one incoming edge and two or more outgoing edges.

Conversely, a join node is another type of control node that synchronises a number of incoming flows into a single outgoing flow. Here, each incoming control flow must present a token to the join node before the node can offer a single token to the outgoing flow. This is represented by a solid bar with two

or more incoming edges and one outgoing edge. Further, this branching and joining behaviour of and IOD must be properly nested.

A decision node is another control node that represents alternative interaction behaviour. This has one incoming flow and two or more outgoing flows. Here, the outgoing flows are guarded, which gives them a mechanism to accept or reject a token. The edge that is actually traversed is selected based on the evaluation of the guards on the outgoing edges. This alternative behaviour is shown using a diamond shaped symbol and the condition statements are represented within the notation `[ ]`.

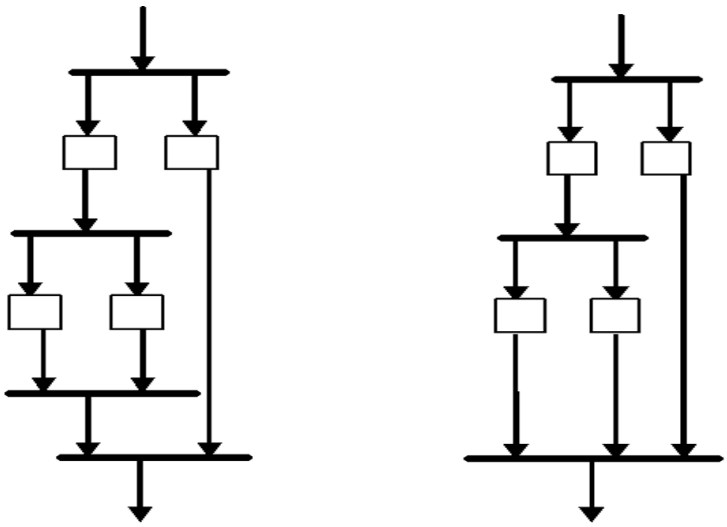
A merge node is another control node that brings together multiple alternative flows and it corresponds to a decision node. It is not used to synchronise concurrent flows, but to accept one among several alternative flows. A merge node has multiple incoming edges and a single outgoing edge, and represents using a diamond shape. In an IOD there is a merge node corresponds to each decision node and they should be properly nested.

Moreover, an IOD may contain constraints expressed within the notation { } to represent semantics such as the time duration between two nodes [Pilone and Pitman, 2005, OMG, 2011a].

Consider the IOD represented in Figure 3.23. The execution of the diagram starts with an interaction use that refers the interactions of the sequence diagram A. This is followed by weak sequencing B with the message m, which is shown as an inline interaction. The time duration between the end of interaction A and the start of interaction B is indicated as $\{0..2\}$, where 0 is the lower bound and 2 is the upper bound. Then there is an alternative behaviour as we find a decision node with constraints on each outgoing edges.

Here, based on the guard condition that evaluates true, either interaction C or D is selected for the execution. Then a merge node brings together

the alternative flows and directed to a fork node. Along that control flow, a parallel execution happens with the interactions E and F. Finally, the flows are joined and direct towards the final node.



(a) (b)

Figure 3.24: The nested behaviour of the control nodes (a) correct (b) incorrect.

Further, an IOD preserves the nested behaviour of the control nodes within its control flow. In other words, for each fork node there is a corresponding join node and for each decision node there is a corresponding merge node. Figure 3.24 (a) and (b), show a correct and incorrect nested behaviour of the control fragments, respectively. In Figure (a) the fork and join nodes are properly nested, whereas in Figure (b) they are not.

3.2.2 Formal Model of an Interaction Overview Diagram

This sub section describes a formal model of an IOD. The definition considers an IOD with the main elements that we use for the model transformations defined in Chapter 6.

We define an IOD as follows:

Definition 3.15 (Interaction Overview Diagram) *An Interaction Overview Diagram I is a structure $I = (N, E, t, l, Exp)$ where*

- *N is a finite set of nodes with two categories: activity nodes and control nodes, such that $N = N_{act} \cup N_{cnt}$;*
  $N_{act} = \{R \cup S\}$ is a disjoint union of set where,
  - *R is a finite set of nodes representing interaction use;*
  - *S is a finite set of nodes representing inline interaction;*

  $N_{cnt} = \{B \cup \{L \cup F \cup D\}\}$ is a disjoint union of set where,
  - *B is a singleton that indicates the initial node;*
  - *L is a finite set of final nodes;*
  - *$F = \{F_{beg}, F_{end}\}$ is a finite set of nodes with parallel behaviour, where F_{beg} is a fork node and F_{end} is the corresponding join node;*
  - *$D = \{D_{beg}, D_{end}\}$ is a finite set of nodes with alternative behaviour, where D_{beg} is a decision node and D_{end} is the corresponding merge node;*
- *E is a finite set of directed edges and may contain a constraint;*
- *$t : E \to (N \times N) \setminus \{(N \times B) \cup (L \times N)\}$ is a total function that assigns a pair of nodes (a source and a target node) to a directed edge; t is not defined for the situations where B becomes the target node and L becomes the source node;*
- *$l : N_{act} \to \mathcal{N}$ is a labelling function, which associates a SD name for an activity node;*

- *The corresponding nodes in F and D are such that*
  - $min : D \rightarrow D_{beg}$;
  - $max : D \rightarrow D_{end}$;
  - $min : F \rightarrow F_{beg}$;
  - $max : F \rightarrow F_{end}$;
  - $c : (F_{beg} \cup D_{beg}) \rightarrow 2^E$ *is a total function which assigns a set of outgoing directed edges to a fork node or a decision node, respectively and* $c : (F_{end} \cup D_{end}) \rightarrow 2^E$ *is a total function which assigns a set of incoming directed edges to a join node or a merge node, respectively: for* $d \in D_{beg}$, $d^{'} \in D_{end}$, $f \in F_{beg}$, $f^{'} \in F_{end}$ *the corresponding cardinalities are same such that* $|c(d) = c(d^{'})|$ *and* $|c(f) = c(f^{'})|$;
  - $r : (D \cup F) \rightarrow 2^{N_{act}}$ *is a function that associates a set of activity nodes to a alternative or parallel behaviour;*
- *Exp is a finite set of expressions such that* $guard : E \rightarrow Exp$ *is a partial function that associates an expression to an edge, where* $t(e) = (d, n)$ *for* $e \in E$, $d \in D_{beg}$, $n \in N \setminus \{B\}$;

An IOD I is described by a set of nodes N and directed edges E that show the control flow between the nodes. For example, consider the IOD shown in Figure 3.25. There are mainly two kinds of possible nodes: activity nodes $N_{act} = \{r_A,\ s_B,\ r_C,\ r_D,\ r_E,\ r_F\}$ that represent the interactions and control nodes $N_{cnt} = \{b, d, d^{'}, f, f^{'}, l\}$ that shows the controlling features such as synchronisation. This distinction can be explicitly referred as $N = N_{act} \cup N_{cnt}$.

The interaction use nodes $r_A, r_C, r_D, r_E, r_F \in R$ and inline interactions $s_B \in S$, nodes are considered as activity nodes and the nodes fork ($f \in F_{beg}$),

join ($f^{'} \in F_{end}$), decision ($d \in D_{beg}$), merger ($d^{'} \in D_{end}$), initial ($b \in B$) and final ($l \in L$) are considered as control nodes. The interactions within the IOD starts with the interaction use r_A followed by the inline interaction s_B. The labelling function l gives the name of the SD, which is referred by a given activity node such that $l(r_A) = A$, $l(s_B) = B$, etc.

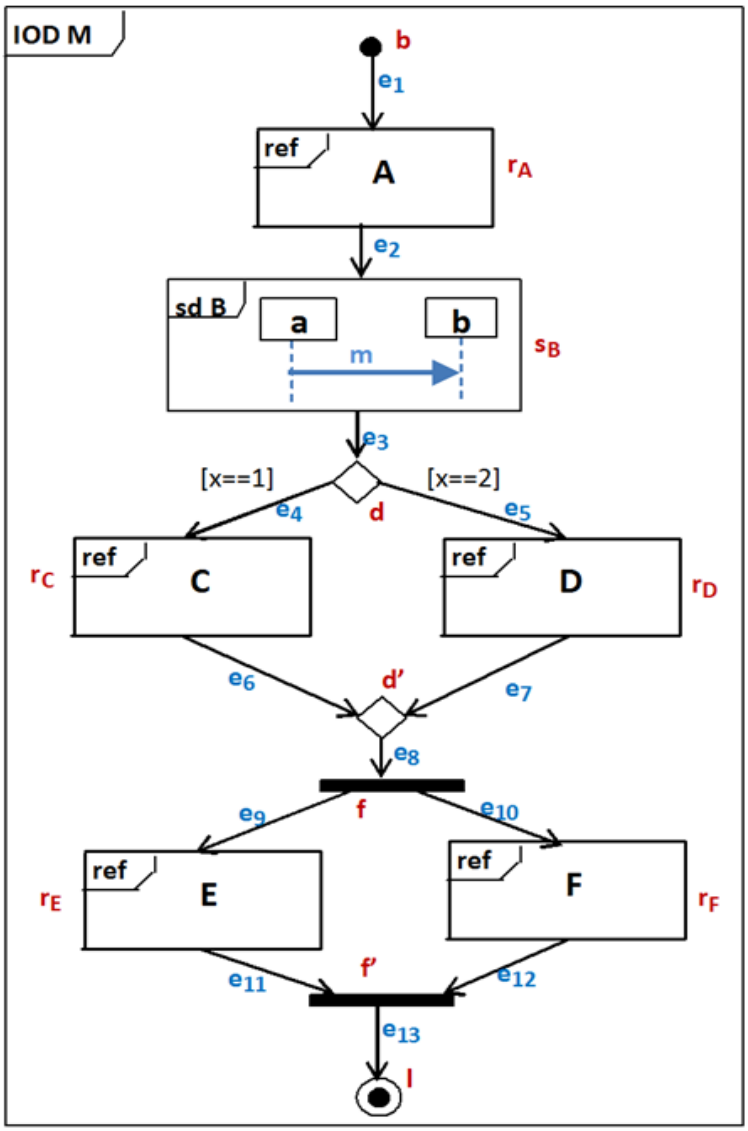


Figure 3.25: An example of an interaction overview diagram.

The directed edges $E = \{e_1, \cdots, e_{13}\}$, link the nodes as defined by the function t. This function returns the source and the target node pair for a given edge, to maintain the control flow of the model. Here, the node B never becomes a target and the node L never becomes a source node. The function t defines the source and target nodes for each edge such that $t(e_2) = (r_A, s_B)$, $t(e_3) = (s_B, d)$, etc.

The function c gives the incoming edges of the join and merge nodes; and the outgoing edges of the fork and decision nodes. For example, outgoing edges of the decision node d is given by the function c such that $c(d) = \{e_4, e_5\}$. Since the control nodes are properly nested, for a given (fork, join) node pair and for a given (decision, merge) node pair, $|c(f) = c(f^{'})|$ and $|c(d) = c(d^{'})|$, respectively for $f \in F_{beg}$, $f^{'} \in F_{end}$ and $d \in D_{beg}$, $d^{'} \in D_{end}$.

An expression is associated with each outgoing edge of the decision node given by the function *guard* in such a way that $guard(e_4) = [x == 1]$. These branch conditions are used in order to distinguish between different possible executions. Here, based on the guard condition that evaluates true, the control flow executes the interaction r_C or r_D and the merge node d' brings together the alternative flows and directs to the fork node f.

The fork operator specifies that the two main paths executed by the system are in parallel; here a parallel execution happens with the nodes r_E and r_F. The join node $f^{'}$ synchronises the control flow and directs towards the final node. Further, for $alt \in D = (D_{beg}, D_{end})$, $r(alt) = \{r_c, r_d\}$ and for $par \in F = (F_{beg}, F_{end})$, $r(par) = \{r_e, r_f\}$. For an IOD I, we write $begin(I) = B$ and $last(I) = L$, which indicates the initial and final node, respectively.

We define an additional function *next* over nodes and directed edges that facilitates to define the transformation rules in Chapter 6. This function returns the *next* edge or node for a given node or edge, respectively.

Definition 3.16 (Function: next) *Let an IOD contains a set of nodes N and edges E. The function next is defined over nodes and edges such that $next : N \cup E \rightarrow 2^{N \cup E}$.*

$$next(x) = \begin{cases} \{e^{'}\} & \Leftarrow x \in N = \{N_{act}, B, F_{end}, D_{end}\} \\ & \text{and } e^{'} \in E \text{ such that} \\ & t(e^{'}) = (x, n_i) \text{ for some } n_i \in N \\ \{e_1, \ldots, e_n\} & \Leftarrow x \in (F_{beg} \cup D_{beg}),\ e_i \in E,\ i \in \mathbb{N} \\ & \text{where } c(x) = \{e_1, \ldots, e_n\} \text{ suchthat} \\ & t(e_i) = (x, n_i) \text{ for some } n_i \in N \\ \{n^{'}\} & \Leftarrow x \in E \text{ and } n^{'} \in N \text{ such that} \\ & t(x) = (n_i, n^{'}) \text{for some } n_i \in N \\ \perp & \Leftarrow x \in L \end{cases}$$

The definition states that the *next* node for an edge is generally given by a singleton containing a node. This node is the target node associated with the edge and can be determined by applying the function t to the edge. For example, in Figure 3.25 $next(e_1) = r_A$, $next(e_2) = s_B$, so on. Also, the next node of an edge can be determined by its target node such that $t(e_3) = (s_B, d)$ and $next(e_3) = d$.

Conversely, the *next* edge for a node depends on its type, if it is a control node. When the node is a fork node or a decision node, the set of next possible edges are given by the union of all outgoing edges such that $next(d) = \{e_4, e_5\}$ and $next(f) = \{e_9, e_{10}\}$. Here, the set of edges can be determined by applying the function c to the corresponding node in F_{beg} or D_{beg} such that $c(f) = \{e_9, e_{10}\}$.

When the control node is a final node L, the function *next* is not defined, as there are no elements after the final node. For all other nodes, there is a unique next edge for a given node such that $next(b) = e_1$, $next(r_A) = e_2$, so on.

Similarly, we can define a function $previous : N \cup E \rightarrow 2^{N \cup E}$, to extend the functions associated with the formal representation of an IOD. Although the definition is not essential for this formal representation, we include the definition as follows for the completion of this section.

Definition 3.17 (Function: previous) *Let an IOD contains a set of nodes N and edges E. The function previous is defined over nodes and edges such that next $: N \cup E \rightarrow 2^{N \cup E}$.*

$$previous(x) = \begin{cases} \{n^{'}\} & \Leftarrow x \in E \text{ and } n^{'} \in N \text{ such that} \\ & t(x) = (n^{'}, n_i) \text{ for some } n_i \in N \\ \{e^{'}\} & \Leftarrow x \in N = \{N_{act}, L, F_{beg}, D_{beg}\} \\ & \text{and } e^{'} \in E \text{ such that} \\ & t(e^{'}) = (n_i, x) \text{ for some } n_i \in N \\ \{e_1, \ldots, e_n\} & \Leftarrow x \in (F_{end} \cup D_{end}),\ e_i \in E,\ i \in \mathbb{N} \\ & \text{where } c(x) = \{e_1, \ldots, e_n\} \text{ suchthat} \\ & t(e_i) = (n_i, x) \text{ for some } n_i \in N \\ \perp & \Leftarrow x \in B \end{cases}$$

Similar to the function *next*, the function *previous* gives the previous node or edge for a given edge or node, respectively. There is a unique previous node for a given edge and the node can be determined as the source node, by applying the function t to the edge. In Figure 3.25, $previous(e_1) = b$ and $t(e_1) = (b, r_A)$.

When the node is any activity node, a fork node, a decision node or a final node; the function *previous* returns a singleton that contains an edge. When it is a join node or a merge node, there may have one or more associated previous edges. In this case, the function *previous* returns the set of previous edges associated with the node such that $previous(d^{'}) = \{e_6, e_7\}$, $d^{'} \in D_{end}$. This can be determined by applying the function c to a join or merge node

such that $c(d^{'}) = \{e_6, e_7\}$. When the node is an initial node, the function *previous* is not defined.

The *next* and *previous* functions over nodes and edges can be described as follows. $next(b) = \{e_1\}$, $next(s_B) = \{e_3\}$, $next(d) = \{e_4, e_5\}$, $next(e_5) = \{r_D\}$, $next(l) = \{\perp\}$ etc. Similarly, $previous(r_C) = \{e_4\}$, $previous(d^{'}) = \{e_6, e_7\}$, $previous\ (e_8) = \{d^{'}\}$, so on.

Further, we can derive chains of interleaved nodes with the use of the function *next*. Consider the IOD shown in Figure 3.25 with nodes $N = \{b$, r_A, s_B, d, r_C, r_D, $d^{'}$, f, r_E, r_F, $f^{'}$, $l\}$ and $E = \{e_1, \cdots, e_{13}\}$. Since the node d and f have more than one next edges, we can derive the valid IOD chains over nodes such as, $b \cdot r_A \cdot s_B \cdot d \cdot r_C \cdot d^{'} \cdot f \cdot r_E \cdot r_F \cdot f^{'} \cdot l$, $b \cdot r_A \cdot s_B \cdot d \cdot r_D \cdot d^{'} \cdot f \cdot r_E \cdot r_F \cdot f^{'} \cdot l$, so on.

In particular, the function *next* uses to define notions of the IOD *chain* as follows. For every edge $e_j \in E$ and for every node $n_i \in N$ in an IOD, since $t(e_j) = (n_{(i-1)}, n_i)$ and $next(e_j) = n_i$ are defined, take n_i and move to the next edge, for $i \in \mathbb{N}$. When a $n_i \in F_{beg}$, all the other nodes that execute in parallel should be taken preserving the order within the chain before the corresponding join node.

The definition 3.18 describes the idea of chains on nodes.

Definition 3.18 (Chain-IOD) *Given an IOD and associated set of nodes N, a chain c is a finite sequence of interleaved nodes such that $c = n_0 \cdot \cdots \cdot n_j \cdot n_k \cdot \cdots \cdot n_f$ where $n_0 \in B, n_f \in L, n_j, n_k \in \{R, S, D_{beg}, D_{end}, F_{beg}, F_{end}\}$; where $e \in E$, $t(e) = (n_i, n_{(i+1)})$ such that $next(n_i) = e$ and $next(e) = n_{(i+1)}$. Further, when $n_j \in F_{beg}$ then for all following nodes that executes in parallel must occur in the chain before $n_k \in F_{end}$.*

In an IOD, the control nodes are a convenience introduced to denote the synchronisation and control flow of the instances within the IOD. These control

nodes do not add anything to the actual words defined over the IOD, that is words are obtained from the activity nodes. Thus, from a chain we can obtain a sequence of nodes n_j over N_{act} as follows: for every node $n_i \in N$, take n_j and move to the next activity node. Hence, the notion of a chain uses to define the notion of IOD *trace*. For example in Figure 3.25, $r_A \cdot s_B \cdot r_C \cdot r_E \cdot r_F$, $r_A \cdot s_B \cdot r_D \cdot r_F \cdot r_E$, etc. can be considered as the traces of IOD_M. We define a trace of an IOD as follows.

Definition 3.19 (IOD-Trace) *A trace of an IOD with set of nodes N_{act} is a possibly infinite word w, $w = n_1 \cdot n_2 \cdot n_3 \ldots$ over the IOD alphabet L_2 iff there exists a chain c of nodes $n_1^{'} \cdot n_1 \cdots n_2^{'} \cdots n_2 \cdots n_n^{'}$ over N where $n_i^{'} \in N_{cnt}$ and $n_i \in N_{act}$ such that w can be derived from c by removing the control nodes.*

Based on the defined formal representation of the IOD, we can define the associated language $L_2(IOD)$. The legal set of traces in an IOD is defined by the control flow of the activity nodes in the IOD. We define L_2 as the alphabet of an IOD over the set of activity nodes N_{act} such that $L_2 = N_{act}$. The associated language $L_2(IOD)$ is defined over the activity nodes for a set of legal traces of the IOD follows.

Definition 3.20 (IOD-Language) *Let a maximal trace be a trace which is not a proper prefix of any other trace. A language of IOD is the set $\mathcal{L}_2(IOD)$ of words over the alphabet L_2, where $\mathcal{L}_2(IOD) = \{W |$ W is a maximal trace of $IOD\}$.*

For the formal representation of the IOD we mainly consider an IOD with only the interaction use nodes and the control flow between the nodes using directed edges. For simplicity we do not consider the tokens associate with the nodes. Further, the defined formal model can be extended with time and stochastic aspects for the future work. Therefore, we do not consider the detail formal model for an IOD, as it is out of scope of this thesis.

3.3 Concluding Remarks

This chapter has given a formal representation for the syntax and semantics of UML2 sequence diagram and possible extension with time and stochastic aspects. Our formal definition is based on UML standards [OMG, 2011a] and we have introduced a number of elements which will facilitate the formal transformation rules given in Chapter 5. Further, this chapter has described interaction overview diagrams that uses sequence diagrams as its nodes and captures the hierarchical view of a set of sequence diagrams.

4 A Formal Model: Coloured Petri Net

As mentioned previously, formal models are important for the specification and analysis of systems. A formalised design model such as the one introduced in Chapter 3 facilitates design specification as its notation is clear, accurate and unambiguous. Design models with a well-defined semantics can be formally verified for correctness and consistency, and hence give us an assurance that the systems we develop behave as expected. For a wide range of systems this is an essential requirement.

This thesis uses coloured Petri nets (CPNs) as the underlying formal model associated with UML 2 sequence diagrams described in the previous chapter. Petri nets (PNs) were first developed by Carl Adam Petri as part of his PhD in 1962 [Petri, 1962]. Since then, PNs have gained much attention and interest both in research with several conference series devoted to them, and in practice with numerous applications. PNs have also been extended in many different ways to capture different kinds of problems more accurately. Notable examples include the Coloured Petri Net (CPN) introduced by K. Jensen [Jensen, 1981, Jensen and Kristensen, 2009, Jensen, 1997a, Jensen, 1994] where colours (essentially denoting types) are assigned to tokens and places, and the Predicate/Transition net introduced by Genrich and Lautenbach [Genrich and Lautenbach, 1981] which is a high-level PN with a set of first order places called predicates. Here, the colours can be used to distinguish between object types and predicates can be used to capture hierarchical relations.

PNs and their extensions have become popular because they combine simple graphical representations with powerful primitives to model concurrency, communication and synchronisation [Hamadi and Benatallah, 2003, Silva and dos Santos, 2004, Chrzastowski-Wachtel et al., 2003, Murata, 1989, Jensen et al., 2007, Billington and Reisig, 1996, Kristensen et al., 2004]. Moreover, PNs have

rich tool support for analysis and simulation [Benatallah et al., 2003,Kristensen et al., 1998,Kounev et al., 2010, Jensen and Kristensen, 2009]. Although, PNs are rich in theory and have been used in practice in several application domains, they cannot replace other currently more popular and informal modelling languages such as UML. Instead they can be used behind the scenes and bring their theoretical results and tools as a supplement to existing modelling languages and methodologies. In other words, we can integrate PNs into commonly used modelling languages with added benefits. Here, we use CPNs because their colour extension is very natural for capturing object types as we will describe in more detail later on.

This chapter describes the syntax and semantics of a CPN in accordance to [Jensen and Kristensen, 2009]. Section 4.1 describes the notions of a CPN and Section 4.1.1 explains in some detail why CPNs are well suited for our needs. Next Section 4.2 presents the theoretical details of a CPN. Section 4.3 describes two possible extensions for the defined CPN model with time and stochastic aspects that facilitate to model real-time and stochastic behaviour of a system, respectively. The final section provides the hierarchical constructs for a CPN that allows composition and decomposition of system models.

4.1 Main Notions of a Coloured Petri Net

A CPN is a directed, connected, bi-partite graph with two node types called *places* and *transitions*. Nodes are connected through directed *arcs* whereby arcs can only connect nodes of different types. A CPN model of a system is both state and action oriented. It describes the sub-states (places) of the system and the operations (transitions) that can cause the model to change state. Graphically, places are represented by circles, transitions by rectangles, and arcs by directed arrows connecting places and transitions or vice versa

[Jensen, 1981, Jensen and Kristensen, 2009, Jensen, 1997a, Jensen, 1994].

Places may contain zero or more tokens, which are usually shown as black dots. A token represents a mark assigned to a place. A token is associated with a data value, which is known as a token colour [Jensen, 1997a, Kristensen et al., 1998] . The colour sets can be constructed using type constructors (more details are given in Section 4.2). There are atomic colour sets such as Boolean, Integer and String, and structured colour sets based on the object instances. A CPN can use different colour tokens to distinguish, for example the occurrence of the same set of actions by different users. Moreover, each place has an associated colour type to determine the kind of data that the place may contain. The associated initial markings of the places describe the objects associated with the system flow and define the initial configuration of the system by indicating how many tokens of different types are available.

Each token carries a data value in a given type that may enable a transition to fire. A transition is enabled, when all the required tokens are available in each input place that leads to the transition. When an enabled transition fires, one token is removed from each input place, and passed onto each output place associated with the transition [Murata, 1989]. Thus, the firing of a transition results in a state change for the tokens.

Places and transitions constitute the net structure together with directed arcs. An arc always connects a place to a transition or a transition to a place. Since the formal model described in this thesis considers both data and control flow, there is a corresponding place for each object state in the model. Therefore, the defined CPN model allows only one token to pass through a place at a time and does not allow multiple arcs between same pair of nodes, which used to indicate multiple tokens that consumed or produced. Moreover this thesis extends the notion of arcs with inhibitor arcs [dos S. Soares and

Vrancken, 2008, Yang et al., 2010, Heiner et al., 2007]. An inhibitor arc connects a place to a transition, and is represented by an arc terminated with a small circle (instead of an arrow in an ordinary arc). When an inhibitor arc connects to a transition, the presence of a token in the input place disables the firing of the transition. The use of inhibitor arcs are described in Chapter 5 when performing the transformations for the negation operator in a sequence diagram.

Additionally, a transition or an arc may have an associated guard (a Boolean expression) to represent system interactions such as the execution of a conditional statement. The guard is required to evaluate true, to enable the binding and to fire the transition. Further, for a system modelled using a CPN, the transitions are the active part of the system and the places are the passive part.

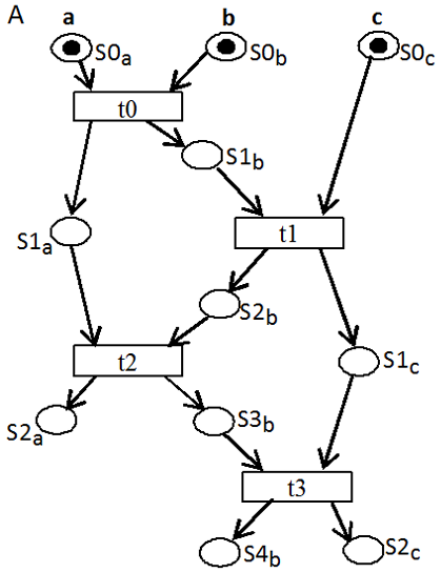


Figure 4.1: An example of a CPN.

Consider the basic CPN shown in Figure 4.1. The CPN named A contains three colours or object types a, b and c, and one token for each object type

initially in places s_{0a}, s_{0b} and s_{0c} respectively. Furthermore, each place in the net has a unique object type and can only contain a token of the same type. CPN_A contains four net transitions $t0$, $t1$, $t2$ and $t3$. The tokens required for each of the transitions to fire indicates the object type involved in the interaction (transition). Directed arcs link the places and transitions to show the control flow of the modelled system. For instance, transition t_1 can only be enabled after transition t_0 has fired as it will move the token of object type b to place s_{1b}.

4.1.1 Motivation for Coloured Petri Nets

As any Petri net, CPNs form a graphically and mathematically defined modelling language appropriate to capture the behaviour of a wide range of systems [Jensen, 1998]. In a CPN, each token has a type called the *token colour* that allows object types and data manipulations [Jensen, 1997a]. Also, with CPNs it is possible to make hierarchical descriptions [Billington, 2004]. Therefore, CPNs combine the capabilities of ordinary Petri nets and high-level features when modelling systems. Since, CPNs have a clearly defined syntax and mathematical semantics, a CPN can be formally verified and checked against dynamic properties that the system it represents should or should not satisfy [Jensen et al., 2007, Mallet et al., 2006, Uzam, 2004, Bernardi et al., 2002].

Generally, PNs with additional features such as colour, time and stochastic aspects are called high-level Petri nets [van der Aals, 1994, Billington, 2004]. The notion of a colour in a CPN can be used to distinguish between types (e.g., object types) of places or tokens. Theoretically, CPNs and PNs have comparable expressive power and as such there is no considerable gain from using CPNs instead of PNs. Nonetheless, the added colours in CPNs make models more natural and better structured [Jensen et al., 2007, Jensen and

Kristensen, 2009]. Ordinary Petri nets have no colours and no mechanisms for adding structure or hierarchies to the nets. This means that a PN can be understood as having only one kind of token and a flat net structure. By contrast, CPNs are capable of distinguishing different (variable and object) types as needed in object-oriented system modelling. The variables in a CPN support the data manipulation between instances. CPNs combine the strengths of PNs (i.e., modelling primitives for resource-sharing, concurrency, communication and synchronisation) with the strengths of programming languages [Kristensen et al., 1998, Jensen, 1994, Jensen and Kristensen, 2009].

Moreover, Petri-nets lack a structuring concept, which makes it difficult to split large models into parts (using either top-down or bottom-up approaches) and hard to reuse. CPNs have a hierarchical structuring mechanism [Kristensen et al., 1998, Jensen and Kristensen, 2009], which we describe in more detail in Section 4.4. Additionally, CPNs can be enriched with timing and stochastic concepts to represent the time taken to execute events and the rate associated with executions, respectively. More on such extensions with timed coloured Petri nets (TCPN) [van der Aalst, 1993, Jensen and Kristensen, 2009] and stochastic coloured Petri nets (SCPN) [Haas, 2002, Zimmermann, 2008] will be described in Section 4.3.

CPNs are aimed at a broad range of systems including embedded systems (to analyse real-time and parallel properties), protocol specification (to congestion control, protocol validation, etc.), manufacturing systems (to analyse failures), business processes (to analyse functional and time properties), and railway systems (to prevent train collision) [Jensen et al., 2007, Kristensen et al., 2004, Vanit-Anunchai, 2010]. Particularly, CPNs can model multiple, independent and dynamic entities in concurrent systems [Jensen, 1998, Reisig et al., 1985, Billington and Reisig, 1996].

As a consequence of a rich and well-defined theory, the token flow within the CPN can be easily simulated to imitate the dynamic and concurrent operations of the modelled system. Through simulation we can investigate different scenarios in a CPN and explore the behaviour of the modelled system by following the behaviour of the tokens across the places in the net when transitions fire. The simulation of a CPN highlights the states of the system (places) and the events (transitions) that cause the system to change state. Hence, the simulation of CPN models can be very effective to analyse system properties such as reachability and liveness. Also formal verifications can be applied to the CPN model to check for performance, consistency and correctness [Kristensen et al., 1998, Jensen and Kristensen, 2009].

4.2 Formal Definition

This section presents a mathematical definition of CPNs in accordance to [Jensen and Kristensen, 2009]. Our definition of a CPN deviates slightly from the original definition given in [Jensen, 1994, Jensen and Kristensen, 2009] and is adaptable for our purpose of modelling inter-object communication. More details are given later in this section. In the following assume the set of diagram names $\mathcal{N}$ of CPN names, and let $\mathcal{E}$ be a finite set of environment instance types.

Definition 4.1 (CPN Formal Model) *A coloured Petri net of name* $d \in \mathcal{N}$ *is defined by a tuple* $CPN_d = (\Sigma, P, T_n, M, l, A, node, m, c, X, Exp, status)$ *where*

- Σ *is a finite set of object colours and* $\Sigma^+ = \Sigma \cup \mathcal{E}$ *includes the environment colours;*
- P *is a finite set of places:*

- T_n *is a finite set of net transitions such that* $T_n = T_n^l \cup T_n^{\neg l}$*;*
- M *is a finite set of labels;*
- $l : T_n \rightarrow M$ *is a partial labelling function, which if defined associates a label from* M*;*
- A *is a finite set of arcs and* $A^+ = A \cup A_{in}$ *includes the inhibitor arcs such that* $A \subseteq (P \times T) \cup (T \times P)$ *:* $P \cap T_n = P \cap A = T_n \cap A = \emptyset$*;*
- $node : A \rightarrow (P \times T_n) \cup (T_n \times P)$ *is a node function, which maps an arc to a pair place-transition or transition-place and* $node : A_{in} \rightarrow (P \times T_n)$*;*
- $m : P \rightarrow \mathbb{N}$ *is the initial marking function associating an initial number of tokens with each place* $p \in P$ *of the net;*
- $c : P \rightarrow \Sigma^+$ *is a colour function associating a colour to a place in the net;*
- $X = \{X_s\}_{s \in \Sigma}$ *is an* Σ*-indexed family of sets of local variables such that* $type(x) \subseteq \Sigma$*;*
- Exp *is a set of expressions such that* $guard : T_n \cup A \rightarrow Exp$ *is a partial function which associates an expression (guard) to a net transition or an arc;*
- $status : P \rightarrow 2^{\{complete, incomplete, safe, unsafe\}}$ *is a function associating a status to a place in the net. By default* $\forall p \in P$*,* $status(p) = \{complete, safe\}$*;*

According to the CPN Definition 4.1, colours, places, transitions, labels, arcs, variables and expressions indicate the elements in the net, whereas the labelling, node, marking, colour and status functions denote the associations and properties of the elements in a CPN. The object or environment types in

a CPN are determined by a colour set Σ^+. Separately, the object colours are represented by Σ and the environment colours are represented by $\mathcal{E}$. Hence, each type has at least one element in the colour set Σ^+. Here, the colours are used to distinguish between the (object or environment) instances involved in the interaction.

A CPN consists of finite sets of places, transitions and arcs, denoted by sets P, T_n, A, respectively. The sets are pairwise disjoint. There is a colour in Σ^+ for each place used in the CPN determining the underlying instance associated with the place. The colour function c maps each place p, to a type $c(p)$. The idea is that each place corresponds to a unique object and thus the colour of a place can be used to identify the type of the object. In this formal model, $min_i(p)$ denotes the minimal (first) place of the colour i and $max_i(p)$ to denote the maximal (last) place of colour i inside the CPN. Moreover, the function *status* applying on a place gives its status. The default statuses of a place are *complete* and $safe$. The notion of status is useful when performing analysis over a CPN model.

There are two kinds of possible net transitions: transitions with a label given by the labelling function l, and transitions without labels. This distinction is also made explicit by assuming disjoint sets $T_n = T_n^l \cup T_n^{\neg l}$. The usage of labelled and unlabelled transitions is further explained in Chapter 5, when defining the transformations from a sequence diagram to a CPN. Generally, net transitions are labelled by the operation name that corresponds to the message label in a sequence diagram. Sometimes net transitions are labelled by diagram names instead. Diagram names are used to introduce structure into a CPN into what is called a hierarchical CPN (cf. Section 4.4). Transitions without labels are a convenience introduced to impose synchronisation of object instances within a CPN and this will be described in detail when

defining transformations in Chapter 5.

Moreover, there are two types of possible arcs: ordinary arcs that pass tokens between places and transitions, and inhibitor arcs that do not pass tokens. The function *node* maps each arc into a pair, where the first element is the source node and the second is the destination node. The two nodes have to be different kind such that one must be a place while the other a transition. That is, the arcs link places to transitions or transitions to places as defined by the function *node* and describe the control flow within the CPN. Notice also that the definition as given here does not allow there to be several arcs between the same ordered pair of nodes. Hence, the set of arcs can be defined as a subset of the place-transition ordered pairs, such that $A \subseteq (P \times T) \cup (T \times P)$.

The function m specifies the initial marking of the places in the net. The number of tokens associated with each place is given by $m(p)$. The colour of the tokens that pass through a given place has the same colour of that place, that is $c(p)$.

Further, transitions and arcs may have guards (Boolean expressions) defined by function *guard* and the expressions use local variables defined in X. For each variable used in the CPN, there is an associated type in Σ. Additional functions, $type(x)$ denotes the type of a variable, $val(x)$ gives the value of the variable, $var(exp)$ denotes the set of variables in an expression and $type(exp)$ indicates the Boolean type, which contains the elements {true, false} and having standard operations. The implementation describes in Chapter 8 uses these functions for evaluating an expression $exp \in Exp$, and acquiring the associated variables $x \in var(exp)$ with $type(x)$.

The given Definition 4.1 for the syntax of a CPN differs slightly from the standard definition [Jensen, 1994,Jensen and Kristensen, 2009]. Our definition of a CPN has been adapted to consider only what is needed when modelling

object interactions. Consequently, initialisation function and parallel arcs between nodes have been removed. Additionally, the colour set extends with environment colours and arcs extend with inhibitor arcs. Also, the definition includes a set of labels M, a set of variables X and a set of expressions Exp. The formal representations given in this definition facilitate to both system design and analysis phases.

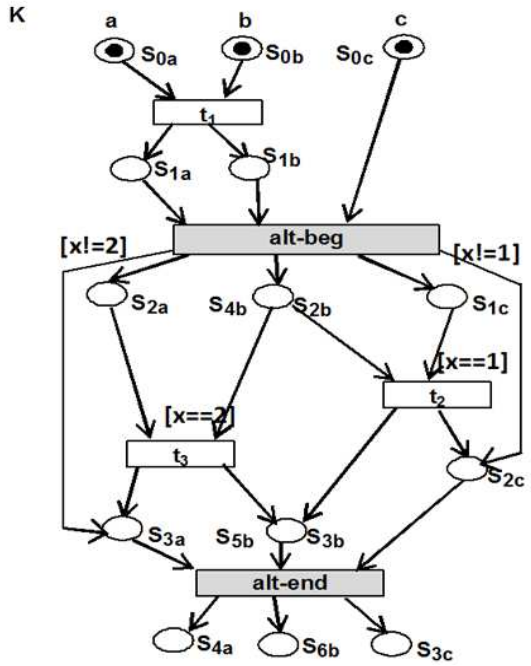


Figure 4.2: A CPN with labelled and unlabelled transitions.

Further, the behaviour underlying a CPN can be obtained from a notion of chain or sequence of execution. A chain in a CPN shows the control flow of a model as an interleaved places and net transitions. Interleaving means the merging of two or more chains such that the occurrences from different chains may come in any order in the resulting chain, while occurrences within the same chain retain their order. This can be obtained for each object involved in the interaction and derived using the function *node* as expected. If no

concurrency or alternative behaviour is present, there is only one valid chain for the CPN model. For example, from the CPN model in Figure 4.1 we obtain the following sequence of places and net transitions for the object b: $S0_b \cdot t0 \cdot S1_b \cdot t1 \cdot S2_b \cdot t2 \cdot S3_b \cdot t3 \cdot S4_b$.

Similarly, consider Figure 4.2 that represents the alternative behaviour with conditional expressions. The instance b contains two chains: $S0_b \cdot t_1 \cdot S1_b \cdot t_{alt-beg} \cdot S2_b \cdot t_2 \cdot S3_b \cdot t_{alt-end} \cdot S6_b$ and $S0_b \cdot t_1 \cdot S1_b \cdot t_{alt-beg} \cdot S4_b \cdot t_3 \cdot S5_b \cdot t_{alt-end} \cdot S6_b$. The formalisation of the notion of a chain can be defined as follows.

Definition 4.2 (CPN Chain) *Given a CPN model CPN and associated set of places P, a chain c for $i \in \Sigma$ is a finite sequence of interleaved places and net transitions where $c(p) = i$: $p \in P$ such that $c = p_0 \cdot t \cdot \cdots \cdot p_j \cdot t' \cdot p_k \cdot t'' \cdot \cdots \cdot p_f$ where $p_0 \in min_i(P)$, $p_f \in max_i(P)$, $p_j, p_k \in P$, $t, t', t'' \in T_n$, such that $node(a_1) = (p_0, t)$, $node(a_k) = (p_j, t')$, $node(a_{k+1}) = (t', p_k)$, so on for $a_k \in A : k \in \mathbb{N}$. Further, for an arbitrary unlabelled net transition in a chain with parallel behaviour, say $t_{par-end} \in T_n^{\neg l}$, then for all net transition $t_k \in T_n$ before $t_{par-end}$ must occur in the chain before $t_{par-end}$.*

From a chain we derive the notion of trace as in Definition 4.3. The legal set of traces in a CPN is defined by the execution order of the message labels of the labelled net transitions. For example, from the chain in the CPN model in Figure 4.1 we obtain the following sequence of net transition labels: $m_0 \cdot m_1 \cdot m_2 \cdot m_3$ where $m_k = l(t_k)$.

We assume that the alphabet L_3 of a CPN is defined over the set of net transition labels M, that is, $L_3 = M$.

Definition 4.3 (CPN Trace) *A trace of a CPN is a possibly infinite word w, $w = m_1 \cdot m_2 \cdot m_3 \cdot \ldots$ over the CPN alphabet L_3 iff there exists a sequence of places $p_1 \cdot p_2 \cdot p_3 \cdot \ldots$ over P of the same colour $c \in \Sigma$, a sequence of arcs*

$a_1 \cdot a_1^{'} \cdot a_2 \cdot a_2^{'} \cdot \ldots$ *over* A, *a sequence of transitions* $\sigma = t_1 \cdot t_2 \cdot t_3 \cdot \ldots$ *over* T_n, *and a sequence of labelled transitions* $t_1^{'} \cdot t_2^{'} \cdot t_3^{'} \cdot \ldots$ *over* T_n^l *obtained from* σ *by removing the transitions without labels, such that* $m(p_1) \geq 1$, $node(a_i) = (p_i, t_i)$, $node(a_i^{'}) = (t_i, p_{i+1})$, *and* $l(t_i^{'}) = m_i$ *for all* $i \in \mathbb{N}$.

The notion of a CPN trace ignores the net transitions in the control flow that have no labels. The transitions without labels are a convenience introduced to denote the synchronisation of object instances and Chapter 5 on model transformations describes this in more detail. Since unlabelled net transitions do not add anything to the actual words defined over a CPN they can be ignored.

Consider the CPN model shown in Figure 4.2 that shows a conditional behaviour. The model consists of three object instances with colours $a, b, c \in \Sigma$ and a corresponding set of places for each colour. The labelled net transitions $t2, t3 \in T_n$ are guarded with conditions such that $guard(t2) = [x == 1]$ and $guard(t3) = [x == 2]$, and the firing transition is selected based on the condition that evaluates to true. Two traces, $m_1 \cdot m_2$ and $m_1 \cdot m_3$ for $m_k = l(t_k) \in M$, over the labelled net transitions can be derived from this figure, that represents the alternative behaviour. Further, the transitions $t_{alt-beg}$ and $t_{alt-end}$ are unlabelled net transitions that are used to synchronise the control flow of the model.

The associated language $L_3(CPN)$ of a CPN is described next.

Definition 4.4 (CPN Language) *The language for a CPN is given by the set* $\mathcal{L}_3(CPN)$ *of words over the alphabet* L_3, *where* $\mathcal{L}_3(CPN) = \{W |$ W *is a maximal trace of* $CPN\}$, *where a trace is maximal when it is not a proper prefix of any other trace.*

The notion of traces and languages are used in Chapter 7, when establishing the semantic correctness of the SD-CPN transformation.

4.3 Extensions of Coloured Petri Nets

This section describes two important extensions of CPNs to add real-time and probabilities to a model. The extensions covered here are timed CPN (TCPN) and stochastic CPN (SCNP).

4.3.1 Timed Coloured Petri Net

Time plays an important role in a range of real-time systems where we need to be able to add real-time constraints on its temporal behaviour. For example, the correct functioning of a system may depend on the time taken by certain activities. CPN models defined in Section 4.2 can be extended with a time concept by defining a Timed CPN. Time aspects are used to handle quantitative time and can be added to CPN models specifying the delays on places and the time taken by transitions to fire. Many timed extensions of CPNs have been proposed including Timed CPNs as in [Jensen and Kristensen, 2009]. One common approach to add time to CPNs is by considering the notion of a timed stamp or time value associated with tokens. In general, tokens carry a time stamp supporting a time-driven execution of the model. The time stamp is used to determine the time that a token can or must be consumed by a transition for it to fire [Jensen, 1997a, Jensen and Kristensen, 2009]. Also Timed CPNs, or TCPNs for short, use a global clock where the time values are integer or real.

Our approach here is different. For a CPN as considered in this thesis there is a unique correspondence between each place in the net and the colour of the token allowed in that place. Therefore, instead of attaching timing information on a token, we can attach it directly to a place or a net transition, which is more natural and also simplifies the presentation. The time constraints associated with the places and transitions in a CPN model are used to specify

the delays on each component. Our Definition 4.5 differs from the standard TCPN definition [Jensen and Kristensen, 2009].

Usually, the firing of a net transition for a CPN is instantaneous. By contrast, in a TCPN a transition may be associated with a non-zero time value representing the time it takes to fire the transition. This is done by preventing the tokens from being sent to the outgoing arcs, until the associated time has been reached. When the time is specified in a transition as an interval, the transition can fire at any time that falls within the specified interval. In any case, we assume the existence of a global clock.

Definition 4.5 gives the denotational semantics of a timed CPN that directly extends Definition 4.1 of an untimed CPN with a notion of time over places and/or transitions.

Definition 4.5 *A timed CPN model (TCPN) of name* $d \in \mathcal{N}$ *is a* $CPN_d = (\Sigma, P, T_n, M, l, A, node, m, c, X, Exp, status)$ *and a partial function* $time_{CPN,d} : P \cup T_n \rightarrow R_0^+ \times R_0^+$ *that associates a time interval to a place or a net transition.*

A TCPN is a CPN where the places and transitions may contain a time constraint given by the partial function *time*. In a TCPN, a place with a timed value is called a *timed place*, and a place without is called an *untimed place*. Similarly, a net transition with a time value is called a *timed net transition*, and other net transitions are called *untimed net transitions*. The time value is given as a pair of non-negative real numbers and represents a time interval $[i, j]$ where $i \leq j$. Notice that if $i = j$, the values is an exact time value. When the time values associated with a transition are based on the global clock, the timed transition is enabled to fire only when the clock is within the specified interval.

Figure 4.3 (a) shows a CPN model with both timed and untimed places and net transitions. The timed places and net transitions are associated with

a time constraint given by the function *time*. For example, the net transition $t1$ contains a time constraint $[0, 2]$ such that $time_{CPN,K}(t1) = [0, 2]$. Similarly, place $S3_b$ contains a time constraint such that $time_{CPN,K}(S3_b) = [1, 3]$. It shows that, a token may stay in that place for a duration between 1 and 3 time units determined by the global system clock. Further, this CPN model shows unlabelled net transitions $loop - beg$ and $loop - end$ that are used to synchronise the control flow of object instances (cf. Chapter 5 for more details).

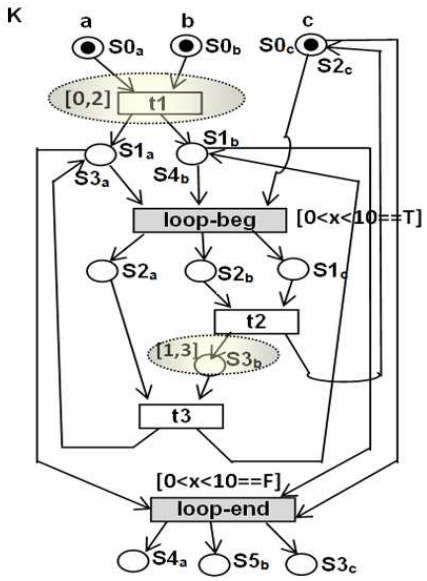


Figure 4.3: A timed coloured Petri net.

In this thesis, the time aspects in a CPN are presented using a non-hierarchical CPN models. However, the definition of timing constructs can be easily generalised to a timed hierarchical CPNs model in a straightforward way.

A timed CPN model enforces additional constraints on the execution of the CPN model compared to the corresponding untimed CPN model. Therefore, a timed CPN model can always be transformed into an untimed CPN model by removing the time constraints associated with places and transitions. That

is, the possible occurrence sequence of a TCPN always form a subset of the occurrence order of the corresponding untimed CPN, such that $TCPN \subseteq CPN$.

The notion of occurrence sequence in a TCPN can be adapted to the notion of CPN chain given in Definition 4.2. Here, we inject the additional time parameters at different places and net transitions to the sequence of interleaved places and net transitions. For example, from the TCPN in Figure 4.3 following chain can be obtained for the object a: $S0_a \cdot (t1, (0, 2)) \cdot S1_a \cdot t_{loop_beg} \cdot S2_a \cdot t3 \cdot S3_a \cdot t_{loop-end} \cdot S4_a$. Similarly, for the notion of trace in a TCPN can be derived by adding additional time parameter to the words in the CPN language. This changes the underlying alphabet accordingly such that, $\mathcal{L}_{TCPN} = M \cup time$.

4.3.2 Stochastic Coloured Petri Net

In a software system design, different design decisions may have a significant impact on the performance of a system. CPNs can be extended with stochastic information that makes it possible to capture the rates associated with activities in the system. Stochastic CPNs can be used for simulation-based performance analysis to measure different performance metrics [Haas, 2002, Zimmermann, 2008].

The CPN model defined in Definition 4.1 can also be extended with stochastic information. This can be done by associating a rate with a net transition in a CPN to represent the rate at which the net transition fires. A rate is always a positive real number determining the negative exponential distribution governing the delay associated with the transition. In contrast, a rate value of zero can be used to block the succeeding net transition. For our model we do not consider such scenarios.

We define a stochastic CPN (SCPN) by introducing a rate labelling function

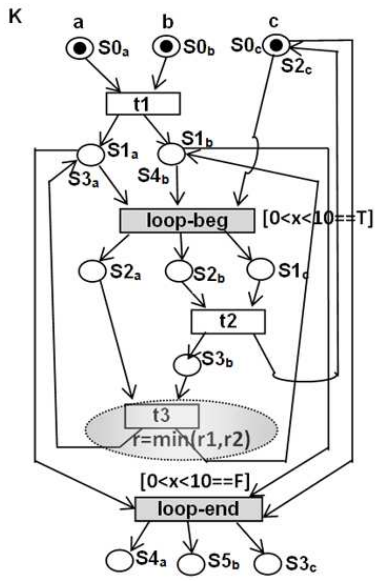


Figure 4.4: A stochastic coloured Petri net

that associates a rate with a given net transition as follows.

Definition 4.6 *A stochastic CPN model (SCPN) of name* $d \in \mathcal{N}$ *is a* $CPN_d = (\Sigma, P, T_n, M, l, A, node, m, c, X, Exp, status)$ *and a partial function* $rate_{CPN,d} : T_n \rightarrow \mathbb{R}^+$ *that associates a rate with a net transition. The rate determines the negative exponential distribution governing the delay associated with the transition.*

A SCPN is a CPN with an additional partial function to add stochastic information to the net. The rate labelling function $rate_{CPN,d}$ associates a rate value, which is a positive real number, with a net transition. In a SCPN, net transitions send the incoming tokens to the outgoing arcs at the rate specified by the function *rate*.

For example, consider the CPN model shown in Figure 4.4 with added stochastic behaviour. The net transition $t3$ is associated with a rate value such that $rate_{CPN,K}(t3) = min(r1, r2)$, where $r1, r2 \in \mathbb{R}^+$. The usual inter-

pretation is that the transition $t3$ fires at a rate that is the minimum of $r1$ and $r2$.

The notion of chain and trace in a SCPN can be adapted to the corresponding notions of a CPN by injecting the additional stochastic parameters at different net transitions to the respective elements. This changes the underlying alphabet of a SCPN accordingly such that, $\mathcal{L}_{TCPN} = M \cup rate$. For example, for the SCPN in Figure 4.4 following trace can be obtained considering the labelled net transitions for the object a: $m_1 \cdot (m_3, (r))$ where $m_1 = l(t1)$ and $m_3 = l(t3)$ and $r = min(r1, r2)$.

A SCPN can always be transformed into a non-stochastic CPN by removing the rate values from the net transitions. Thus, $SCPN \subseteq CPN$. Although the SCPN Definition 4.6 is defined using a non-hierarchical CPN models, these stochastic constructs can be easily applied to a hierarchical stochastic CPN model (HSCPN). We treat hierarchical CPNs next.

4.4 Hierarchical Coloured Petri Net

In practice, a large system cannot be adequately represented by a single CPN model in a way that keeps it is still clear and understandable in overall. These large complex systems brought the need of more powerful structuring mechanisms to handle larger CPN models and resulted in the concept of hierarchical nets. Generally, high-level abstraction models are constructed in early stages of the design phase and are gradually refined to build a detailed design of the system. This helps to focus on only a few details at a time. In general, hierarchical modelling supports system analysis at different levels of abstraction, showing different views of the system and enabling the reuse of system parts. It makes it easier and more flexible to model a system whilst also making it easier to keep system consistency throughout. Generally, a component

can be modelled without full details from the beginning and ensure the consistency when moving between different abstract levels [Fehling, 1993]. Moreover, model analysis with the modular approach often decreases the complexity of the analysis task. Further, with the use of reusable sub models this can be a cost-effective way to obtain an executable prototype of a system. Hence, hierarchical models have more modelling power.

CPN models support a hierarchical structuring mechanism by introducing so-called subnets or modules [Jensen and Kristensen, 2009]. CPNs with such a mechanism are known as Hierarchical CPNs (HCPN for short). The idea behind the HCPN theory is to allow the construction of a large model by using a number of small CPNs with well-defined interfaces, which are related to each other using well-defined interactions. A module may have sub-modules, and the composition of sub-modules form a new module. This notion of a module is similar to the hierarchical constructs in many graphical description languages such as data flow diagrams and the module concepts in high-level programming languages [Jensen, 1998]. Figure 4.5 shows a hierarchical representation of a set of CPN models, where the sub models are referred by the model in the preceding level. HCPNs are rich with features that enhance the modularity and understandability of a design model and have shown to be adequate to support the modelling of large-scale industrial projects [Benatallah et al., 2003, Thomas et al., 1996, Y.Yang et al., 2005, Elkoutbi and Keller, 1998].

The modular capabilities of HCPNs enable model construction with both top-down and bottom-up design approaches. A hierarchical representation of a system makes it possible to specify a simple description of an operation without considering its internal details. Full details are given at a lower level, and a reference is kept to be able to move between levels. Chapter 6 describes partial and incremental transformation with HCPNs in more detail.

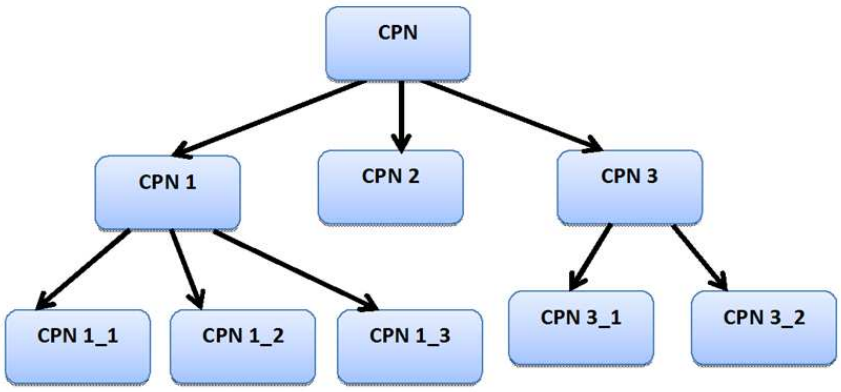


Figure 4.5: Hierarchical view of a set of models.

In a HCPN, a transition may correspond to a complex behaviour given by another CPN. In that case, the top level transition can be seen as an aggregation of places and transitions, which when removed make the HCPN clearer and giving a broader system view. Our Definition 4.7 of a HCPN differs from the original definition given in [Jensen and Kristensen, 2009] and is defined to consider only the inter-model communication with reference and decomposition behaviours. The following definition summarises the semantic concept and notations of a hierarchical CPN. In the following recall that $\mathcal{N}$ denotes the set of all CPN names.

Definition 4.7 (Hierarchical Coloured Petri Net) *A Hierarchical Coloured Petri Net (HCPN) of name $d \in \mathcal{N}$ extends a CPN such that $HCPN_d = (\Sigma, P, T_n, M, l, A, node, m, c, X, Exp, status, r)$ with functions l and r given by*

- *$l : T_n \rightarrow M \cup \mathcal{N} \setminus \{d\}$ is a partial labelling function associating a label from M or $\mathcal{N}$ to a net transition; and*
- *$r : \Sigma \rightarrow \mathcal{N} \setminus \{d\}$ is an object colour reference function, which if defined associates a name to an object colour.*

A HCPN is an extension of a CPN with elements that represent hierarchical behaviour. There are two kinds of structuring mechanisms allowed in our definition. The simplest is given by function l where some of the net transitions are associated with a CPN name where the underlying CPN encapsulates the behaviour of that transition at a lower level. A further mechanism is given by the partial function r, which if defined associates a CPN name with a colour that represents the object with the hierarchical behaviour. Generally, a more detailed system can be modelled by substituting a transition or a colour that associates with the partial function l and r, respectively, that convey the behaviour of the referenced CPN model. These substitutions do not require fundamentally new details and only need to define and establish the proper connections between the relevant nodes in both CPNs. More details on CPN model composition are discussed in Chapter 6.

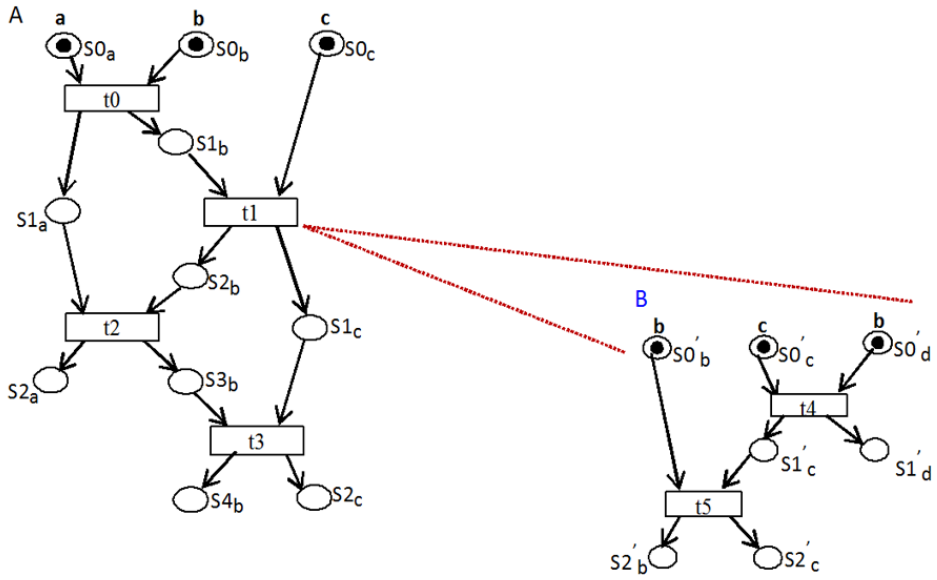


Figure 4.6: $HCPN_A$ and the referred CPN_B where $l(t1) = B$.

Consider the CPN examples shown in Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9. These examples show the construction of the detailed CPN

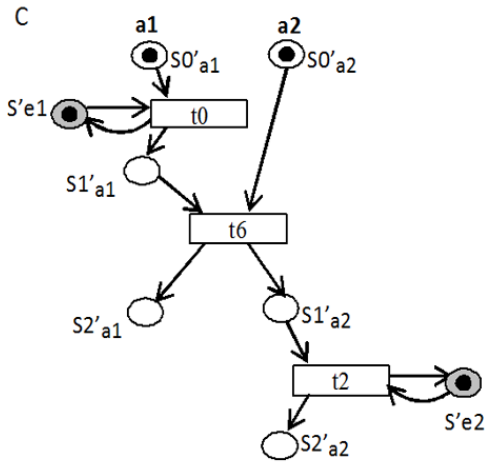


Figure 4.7: CPN_C referred by $HCPN_A$ where $r(a) = C$.

models (1) CPN_{AB}, using $HCPN_A$ and CPN_B, (2)CPN_{ABC}, by referring to $HCPN_A$, CPN_B and CPN_C. The relations between these models are defined using the partial functions l and r as follows. The net transition $t1$ in the model $HCPN_A$ has a reference to model CPN_B with the labelling function $l(t1) = B$. Therefore the transition $t1$ can be substituted by the model CPN_B. Here, the model CPN_B contains a detailed design description of the operation represented by the corresponding substitution transition $t1$.

Further, this thesis defines another decomposition mechanism for CPNs using the object colour reference function. The object a in the model $HCPN_A$ can be decomposed further by associating a CPN model name as a reference to the colour of the object, in such a way that $r(a) = C$. The composition of both CPN_B and CPN_C with the model $HCPN_A$ is shown in CPN_{ABC} in Figure 4.9.

A HCPN can be easily used to construct an equivalent CPN and vice versa. Both models use the same set of places, net transitions, occurrence sequences and they are behaviourally equivalent. Similar to the CPN, the notion of a chain in a HCPN is consist of a set of interleaved places and net transitions.

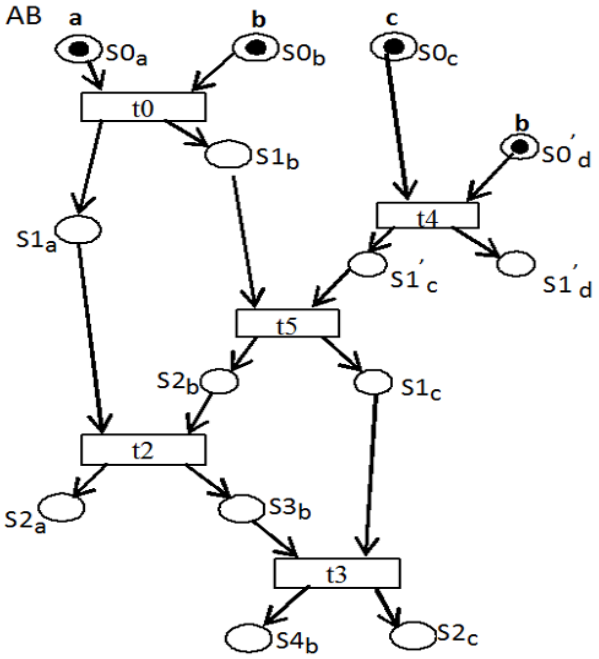


Figure 4.8: CPN_{AB} obtained by $HCPN_A$ and CPN_B.

Additionally, we inject the relevant hierarchical references as parameters at different net transitions and the considered colours, which can be replaced by a chain of the referred CPN.

Consider a $HCPN_A$, colour $a \in \Sigma$ and $r(a) = C$ where C is a name of a CPN. Any chain of places and transitions of colour a are replaced by a more detailed chain of behaviour of CPN_C. Similar procedure is applied for the net transition $t1 \in T_n$ and $l(t1) = B$ where B is a name of a CPN.

Conversely, each CPN name occurrence in a HCPN can be replaced by the corresponding behaviour of the CPN in such a way that if there are no name occurrences left we have a CPN as defined in Definition 4.1.

For example, consider $HCPN_A$ shown in Figure 4.6. For the colour b following chain can be obtained: $S0_b \cdot t0 \cdot S1_b \cdot (t1, B) \cdot S2_b \cdot t2 \cdot S3_b \cdot t3 \cdot S4_b$. Since the net transition $t1$ has a reference to CPN_B, for the colour b in Figure 4.6

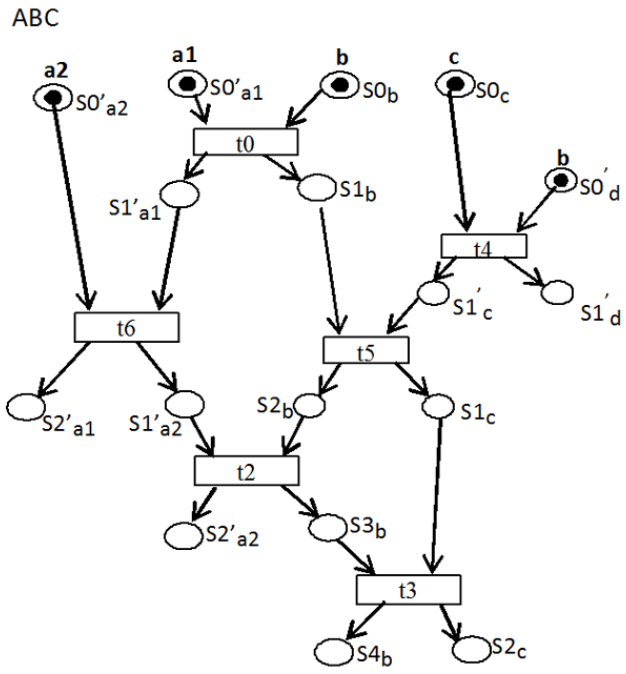


Figure 4.9: CPN_{ABC} obtained by $HCPN_A$, CPN_B and CPN_C.

we can obtain the chain: $S0'_b \cdot t5 \cdot S2'_b$. Now, consider CPN_{AB} in Figure 4.8, which is obtained by substituting $t1$ in $HCPN_A$ by CPN_B. In CPN_{AB}, the chain of colour b is, $S0_b \cdot t0 \cdot S1_b \cdot t5 \cdot S2_b \cdot t2 \cdot S3_b \cdot t3 \cdot S4_b$. Thus, a chain of a HCPN can be reformed to an equivalent chain of a CPN.

Similarly, the notion of trace in a HCPN can be obtained directly by using the same alphabet defined over the set of net transition labels $l(t)$: $t \in T_n$ as in CPN. We can define a HCPN trace over $L3$ as a possibly infinite word $w = m_1 \cdot m_2 \cdot m_3 \ldots$ iff there exists a valid chain c of interleaved places and net transitions for some colour such that we can derive w from c. Here, the colours with the function r and the net transitions with $l(t) \rightarrow \mathcal{N} \setminus \{d\}$ where d is the name of the HCPN, can be replaced by the relevant traces of the corresponding CPNs referred by these functions in order to get more detailed trace. Thus, an equivalent trace as of a CPN (Definition 4.3) can be derived

by replacing the words with the referred traces.

Consider Figure 4.6. The trace of colour c in $HCPN_A$ is given as $B \cdot m_3$ where $B = l(t_1)$ and $m_3 = l(t_3)$. Here, B can be replaced by the corresponding trace of the colour c in CPN_B: $m_4 \cdot m_5$ where $m_4 = l(t_4)$ and $m_5 = l(t_5)$. The resulted trace, as in CPN_{AB} in Figure 4.8, can be derived as $m_4 \cdot m_5 \cdot m_3$. Thus a trace in a HCPN leads to a trace in a CPN.

4.5 Concluding Remarks

This chapter started with a generic introduction to CPNs, and a motivation as to why CPNs are useful for our present needs. The described formal representations for the CPN models are sufficiently complete for the context and the scope of this thesis. The content of the theoretical concepts defined in this chapter are partially based on standard theories behind CPN models [Jensen, 1981, Jensen, 1998, Jensen and Kristensen, 2009, Jensen, 1997a, Jensen, 1994, Thomas et al., 1996, Jensen et al., 2007] and have adapted more closely to our needs in this thesis.

In addition, we have shown extensions of CPNs for timing and stochastic aspects, namely timed CPN (TCPN) and stochastic CPN (SCPN), respectively. These extensions are important because they enable us to use our framework for a wider range of systems and address performance analysis. Further, we described an extension of CPNs with structuring mechanism called hierarchical CPNs that allows modelling and analysis of large and complex systems.

These CPN variants (TCPN, SCPN, HCPN) constitute high-level Petri nets [Billington, 2004] and have extended by associating the required parameters. The advantage of this is to have a uniform language structure for all the variants of CPNs being discussed in this thesis.

5 Model Transformation: Sequence Diagrams to Coloured Petri Nets

Model-driven development (MDD) relies on automated model transformations in the design process [Kleppe et al., 2003, Stahl et al., 2006, Sendall and Kozaczynski, 2003] to assist the rapid adaptation and evolution of models at various levels of detail [Kuznetsov, 2007, Cuadrado et al., 2011]. Model transformations bridge the gap between different models by automating various tasks that keep models consistent and facilitate techniques such as model simulation and/or formal verification. Thus, modelling and transformations are elevated to key artefacts in model-based software development. Hence, there is a demand for researching ways in which model transformation can become more efficient, complete and consistent in software system development. Our interest in particular concerns the link between model-driven development and formal methods.

Performing a model transformation requires a clear understanding of the abstract syntax and semantics of both the source and target models. Chapter 3 has introduced a formal definition and semantics of UML 2 sequence diagrams (SDs), and Chapter 4 has described coloured Petri nets (CPNs) as needed for our purposes. This chapter focuses on a definition of a complete and consistent *exogenous* model-to-model transformation (also referred to as M2M) from SDs to CPNs in such a way that the target model can be analysed and the results of the analysis be lifted back to the source model. All the transformation rules from a SD to a CPN are presented formally following the operational approach and illustrated with examples.

This chapter starts by describing the model transformation framework considered for the thesis. Section 5.2 and Section 5.3 focus on main SD-to-CPN

transformation by addressing all the main constructs. Section 5.4 describes different kinds of interaction fragments available in UML2 SDs and how they are best represented in CPNs.

5.1 Model Transformation Framework

Generally, different views of system models can be transformed to various formal models, which enable model analyses, by defining a precise set of transformation rules. With a definition of a framework, it is much easier to apply model transformations, extensions to other formal models, and reuse the transformations [Bowles and Meedeniya, 2010,Bowles and Meedeniya, 2012b,Bowles and Meedeniya, 2012a].

The work done in this thesis can be seen as part of a more general MDD-based framework to validate UML models using coloured Petri nets, and hence exploit existing coloured Petri net analysis and verification tools for UML-based design. The model transformation framework shown in Figure 5.1 includes rules that transform: (1) UML2 sequence diagrams (SDs) to CPNs, (2) time annotations of SDs to TCPNs, (3) stochastic annotations of SDs to SCPNs, (4) hierarchical annotations of SDs to HCPNs, (5) SDs to IODs, (6) CPNs to hierarchical CPNs, (7) composition of multiple SDs, and (8) composition of multiple CPNs. The approach we take is flexible for extensions. For example, the transformation rules for the main notations can be extended to consider stochastic and real-time behaviour. The flexibility comes from choosing the transformation and target model depending on the intended analysis and verifies the original model.

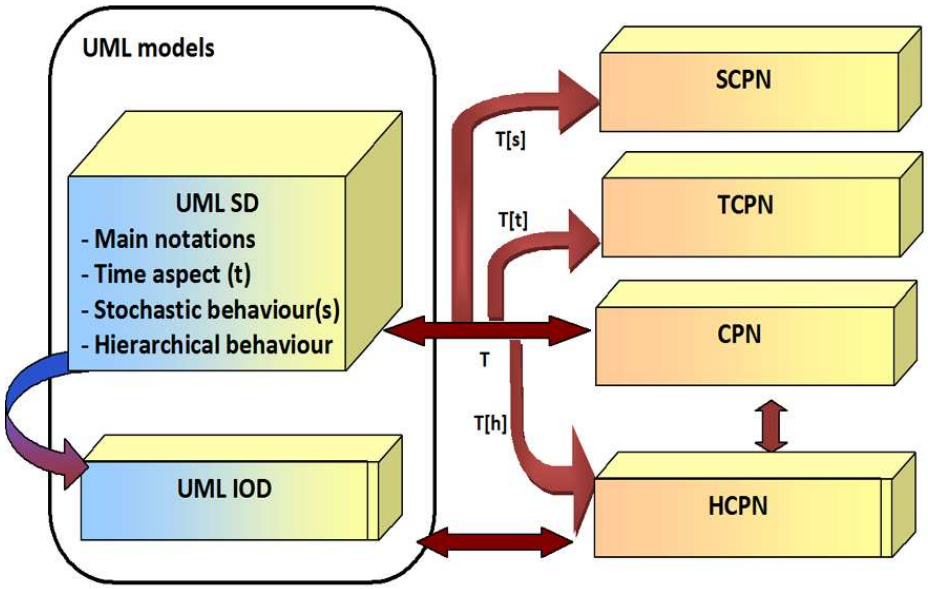


Figure 5.1: The model transformation framework.

5.2 Main Transformation Rules

This section describes the main rules for transforming the essential elements and concepts of a UML sequence diagram into a CPN.

A transformation rule consists of a set of named elements of the source and the target models and a definition of how they are related. Optionally, a rule may contain transformation parameters and constraints that must hold before the rule can be applied.

As defined in Chapter 3, Definition 3.1, a sequence diagram with name d is a tuple $SD_d = (I, E, <, M, T, F, ref, X, Exp)$, consisting of a set of object instances I (with I^+ including environment instances), a set of events E, a partial order over events $<$, a set of message labels M, a set of local transitions T, a set of interaction fragment identifiers F, a function ref for interaction uses (partially defined over object instances and fragment identifiers), a set of variables X and expressions Exp.

As described in Chapter 4 Definition 4.1, a CPN of name d is defined by a tuple $CPN_d = (\Sigma, P, T_n, M, l, A, node, m, c, X, Exp, status)$, consisting of a set of object colours Σ (with Σ^+ including environment colours), a set of places P, a set of net transitions T_n, a set of labels M, a labelling function l defined over net transitions, a set of arcs A and a *node* function matching arcs to pairs of places and net transitions, an initial marking for places given by m, a colour function c over places, a set of variables X, expressions Exp, and the status of each place.

In the following let SD_d be a sequence diagram named d, with associated set of state locations given by S, and the associated CPN_d given by our transformation τ, where $(\tau(SD_d) = CPN_d)$ be a coloured Petri net with name d (see Figure 5.2). The basic transformation rules for the essential elements of a SD to a CPN are defined below.

The target CPN has the same name as the source SD.

Rule 5.1 *Let SD_d be a sequence diagram with name d. The CPN obtained by transformation τ, $\tau(SD_d) = CPN_d$ has the same name d.*

The (object and environment) instances in a SD_d are transformed into matching colours in the corresponding CPN_d.

Rule 5.2 *For all instances in a SD there exists a corresponding colour in CPN: $\forall_{i \in I^+} \exists'_{o \in \Sigma^+}$ where $\tau(i) = o$.*

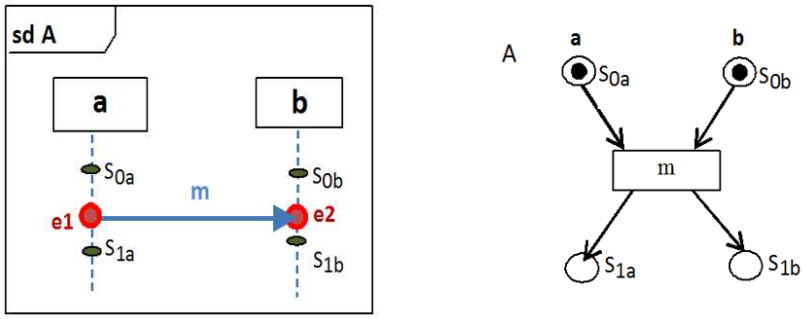
The state locations that belong to an instance are transformed into places in the CPN, such that the colour of the place matches the instance type.

Rule 5.3 *For all state locations of instances in a SD there exists a corresponding place in the CPN with the colour of the underlying instance: $\forall_{s \in S^i, i \in I^+}, \exists_{p \in P}$: $\tau(s) = p$ and $c(p) = \tau(i)$.*

Notice that in Rule 5.3 above, we do not require a unique correspondence between state locations and places. Several state locations may be mapped onto the same place as will be described in Section 5.4.

When transforming state locations into places, only the places corresponding to initial state locations have a defined initial marking. We assume an initial marking of one token per place, but this could be changed by explicitly adding an annotation to the SD, which would lead to a new rule (not given here).

Rule 5.4 *For all initial state locations in a SD the corresponding places in the CPN have a defined initial marking set to 1:* $\forall_{s \in S^i_{ini}, i \in I^+} : m(\tau(s)) = 1$*, where* $\tau(s) \in P$*. Places associated with non-initial state locations have no initial marking.*



(a) (b)

Figure 5.2: A sequence diagram with a local transition (a) and the corresponding CPN (b).

Consider the SD and the corresponding CPN shown in Figure 5.2. The model CPN_A is obtained by applying Rule 5.1 and Rule 5.6 to SD_A.

The state locations in the SD (which interleave with the events) have a direct mapping with the places in the CPN. We use the same alphabet in both models to make this correspondence obvious. The same applies to instance

names and colours in the CPN, which are graphically indicated next to the initial place of that colour.

The execution of a local transition in a SD is represented as the firing of a net transition in the corresponding CPN. The SD consists of two object instances $a, b \in I$ and the CPN has corresponding colours $a, b \in \Sigma$. The initial state locations $S_{0a}, S_{0b} \in S_{ini}$ are mapped onto the places $S_{0a}, S_{0b} \in P$: $c(S_{0a}) = a$ and $c(S_{0b}) = b$, with initial marking given by $m(S_{0a}) = 1$ and $m(S_{0a}) = 1$. Also $t(S_{1a}) = S_{1a} \in P$ and $t(S_{1b}) = S_{1b} \in P$

The next rule states that each local transition in a SD is mapped onto a corresponding net transition in the CPN. Here the previous and next state locations of the events for the instances involved in the local transitions are mapped onto places with corresponding colours in the CPN, besides arcs link places and the net transition as expected using the function *node*. The functions *next* and μ are used to derive the relationship between the state location and the associated event for a given instance (cf. Chapter 3). In the sequel, we omit τ in the expressions provided it is clear from the context what we mean.

Rule 5.5 *Let* $e \in E_i$ *and* $e^{'} \in E_j$ *with* $i, j \in I$, $t \in T$ *with* $t = (e, m, e^{'})$, $e \in next_i(s_{0i})$, $e^{'} \in next_j(s_{0j})$, $\mu_i(m, e) = s_{1i}$ *and* $\mu_j(m, e^{'}) = s_{1j}$. *There is a unique matching of local transitions and net transitions:* $\forall_{t \in T} \exists'_{t' \in T_n^l}$, *and* $\exists_{a_{0i}, a_{0j}, a_{1i}, a_{1j} \in A}$: $l(t^{'}) = m$, $c(s_{0i}) = c(s_{1i}) = i$, $c(s_{0j}) = c(s_{1j}) = j$, *and* $node(a_{0k}) = (s_{0k}, t^{'})$ *and* $node(a_{1k}) = (t^{'}, s_{1k})$ *for* $k \in \{i, j\}$ *and* $i, j \in I^{+}$.

The message names of the interactions in a SD are mapped to the net transitions labels in the corresponding CPN. In particular, we use the same set of labels M in both the SD and CPN definitions.

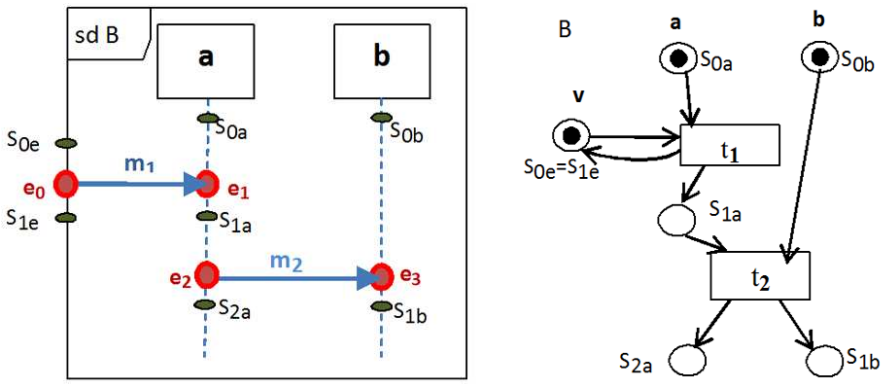
Rule 5.6 *Let $m \in M$ be a message label associated with a local transition $t = (e, m, e^{'}) \in T$ in the SD, then the corresponding net transition $\tau(t) = t^{'} \in T_n$ in the CPN has $l(t^{'}) = m$.*

In Figure 5.2, take the local transition $t = (e_1, m, e_2)$ with $\mu_a(m, e_1) = S_{1a}$, $\mu_b(m, e_2) = S_{1b}$, $next_a(S_{0a}) = \{e_1\}$ and $next_b(S_{0b}) = \{e_2\}$. The corresponding CPN contains a matching transition $t^{'} \in T_n^l$ with the same label, i.e., $l(t^{'}) = m$. The internal state locations given by the function μ are mapped to places $S_{1a}, S_{1b} \in P$ where $c(S_{1a}) = a$ and $c(S_{1b}) = b$. The associated arcs $a_{0a}, a_{0b}, a_{1a}, a_{1b} \in A$ connect the places and transitions as given by the function *node*. I.e. $node(a_{0a}) = (S_{0a}, t^{'})$, $node(a_{0b}) = (S_{0b}, t^{'})$, $node(a_{1a}) = (t^{'}, S_{1a})$, $node(a_{1b}) = (t^{'}, S_{1b})$.

As seen earlier, the interactions in a SD may involve environment instances given by the set *Env* with $I^+ = I \cup Env$ and $I \cap Env = \emptyset$. As described in Chapter 3, environment instances are involved in interactions through gate events. In a local transition either the source or the target event may be associated with an environment instance, but never with both at the same time. Each environment instance has a unique start and end state location, but do not have internal state locations.

When transforming a local transition with a gate event at either end, we impose an equality between the corresponding places of the CPN: i.e. the places that correspond to the initial and end environment state locations are matched. Environment instances are reduced to having one place only in a CPN.

Rule 5.7 *Let $t \in T$ with a gate event $e^{'}$ such that $t = (e, m, e^{'})$ or $t = (e^{'}, m, e)$, where $e^{'} \in next_j(s_{0j})$, $\mu_j(m, e^{'}) = s_{1j}$, and $j \in Env$. In addition to Rule 5.5, $\tau(s_{0j}) = \tau(s_{1j}) \in P$ and $c(\tau(s_{0j})) = c(\tau(s_{1j})) \in \mathcal{E}$.*

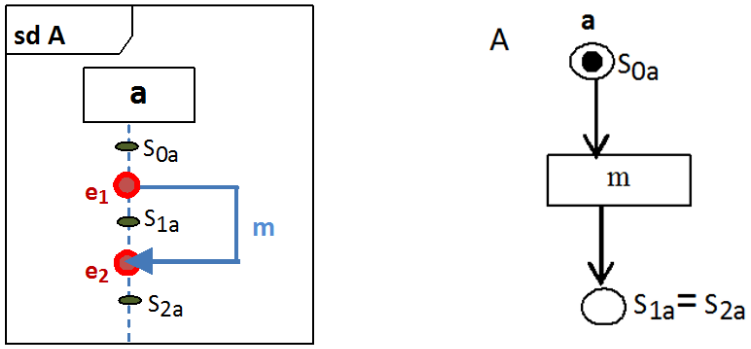


(a) (b)

Figure 5.3: A sequence diagram with an environment instance (a) and the corresponding CPN (b).

To illustrate the transformation of a local transition with a gate, consider the SD and the corresponding CPN shown in Figure 5.3. SD_b consists of object instances $a, b \in I$ and the environment instance $v \in Env$. The interactions start with message $m1$ being sent (by the environment) and being received by instance a, where the sending of $m1$ is a gate. This triggers message m_2 being sent from a to b. By applying the basic transformation rules, the corresponding CPN contains colours $a, b \in \Sigma$ and $v \in \mathcal{E}$.

Consider the interaction of the local transition $t_1 = (e_0, m_1, e_1)$ associated with the environment instance. Here, $e_0 \in next_v(S_{0e})$ and $\mu_v(m_1, e_0) = S_{1e}$. Consequently, CPN_B contains a matching net transition t_1 with label $l(t_1) = m_1$, and places $S_{0e} = S_{1e}$ (since $v \in \mathcal{E}$), $S_{0a}, S_{1a} \in P$ where $c(S_{0e}) = c(S_{1e}) = v$ and $c(S_{0a})c(S_{1a}) = a$. In the CPN representation, there is a unique matching of the places associated with the environment state locations of t_1 determined by $S_{0e} = S_{1e}$. The CPN also must contain arcs connecting the places and the transition as expected, in such a way that, for $a_{0e}, a_{1e}, a_{0a}, a_{1a} \in A$ with $node(a_{0e}) = (S_{0e}, t_1)$, $node(a_{0a}) = (S_{0a}, t_1)$, $node(a_{1e}) = (t_1, S_{1e})$, $node(a_{1a}) = (t_1, S_{1a})$.



(a) (b)

Figure 5.4: A sequence diagram with a self-transition (a) and the corresponding CPN (b).

Local transitions can have the same sender and receiver in which case they are known as self-transitions. In a self-transition, the sender and the receiver events are partially ordered as expected. Rule 5.8 defines the transformation of a self-transition in a SD, to the corresponding CPN representation.

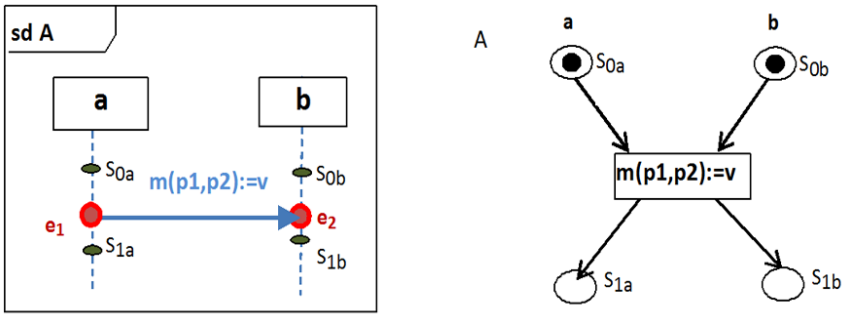
Rule 5.8 *Let $t \in T$ be a self-transition for instance i such that $t = (e_n, m, e_{(n+1)})$, where $e_n < e_{(n+1)}$, for $e_n, e_{(n+1)} \in E_i$ and $\mu_i(m, e_n) = s_{1i}$, $e_{(n+1)} \in next_i(s_{1i})$ and $\mu_i(m, e_{(n+1)}) = s_{2i}$. There is a unique matching of the places associated with the state locations of t determined by μ: $\forall_{t \in T}$, $\exists'_{t' \in T_n}$, $\exists_{s_{1i}, s_{2i} \in P}$, $\exists_{a_{1i} \in A}$, : $\tau(t) = t'$, $l(t') = m$, $c(s_{1i}) = c(s_{2i}) = i$, $node(a_{1i}) = (t', s_{2i})$ and $s_{1i} = s_{2i}$.*

To illustrate the transformation of a self-transition, consider the SD and the corresponding CPN shown in Figure 5.4. The instance $a \in I$ in SD_A is involved in a self-transition $t = (e_1, m, e_2)$, where $e_1 < e_2$. Here, $e_1 \in next_a(S_{0a})$, $\mu_a(m, e_1) = S_{1a}$ and $\mu_a(m, e_2) = S_{2a}$. The corresponding CPN_A contains a colour $a \in \Sigma$, and a matching net transition $t' \in T_n$ with $l(t') = m$ and places $S_{0a}, S_{1a}, S_{2a} \in P$. The self-transition is transformed in a way that the places associated with the state locations given by μ are equivalent, such that

$S_{1a} = S_{2a}$.

As explained in this section, when a local transition in a SD is transformed to a net transition in the CPN, the net transition is labelled by the corresponding message label of the local transition. However, a net transition is labelled not by a message label but by a diagram name when representing a reference behaviour (ref). Such situations occur when a net transition is in effect composite transition that convey the behaviour of a referenced CPN and will be described in Chapter 6.

Also, there are net transitions without a labelling function and these net transitions are a convenience introduced to impose synchronisation of object instances when entering and leaving an interaction fragment in a SD. Section 5.4 explains this behaviour.



(a) (b)

Figure 5.5: A sequence diagram with local variables (a) and the corresponding CPN (b).

Further, SDs may contain local variables. For example, if a transition carries a return value, that value is saved to a variable in the model. The local variables and expressions in a SD are transformed into matching variables and expressions in the corresponding CPN. The following rules define the transformations of local variables and expressions from a SD to a CPN.

Rule 5.9 *For a variable $x_1 \in X_{SD_d}$ in a SD_d there exists a corresponding variable $x_1^{'} \in X_{CPN_d}$ and a colour in $o \in \Sigma$ in CPN_d obtained by transformation τ. Here, $\tau(x_1) = x_1^{'}$ and $type(x) = o$.*

Rule 5.10 *For all expressions $exp \in Exp_{SD_d}$ in a SD_d there exists a corresponding expression $exp^{'} \in Exp_{CPN_d}$, where CPN_d obtained by transformation τ and $\tau(exp) = exp^{'}$.*

Consider the SD and the corresponding CPN shown in Figure 5.5 with local variables. The local transition $t = (e_1, m, e_2)$ in SD_A contains local variables p as a parameter that passes to the receiver and variable r as a return value that obtains by sending the message m. The corresponding CPN_A contains the matching local variables $p^{'}, r^{'} \in X_{CPN_A}$ obtained by $\tau(p) = p^{'}$ and $\tau(r) = r^{'}$.

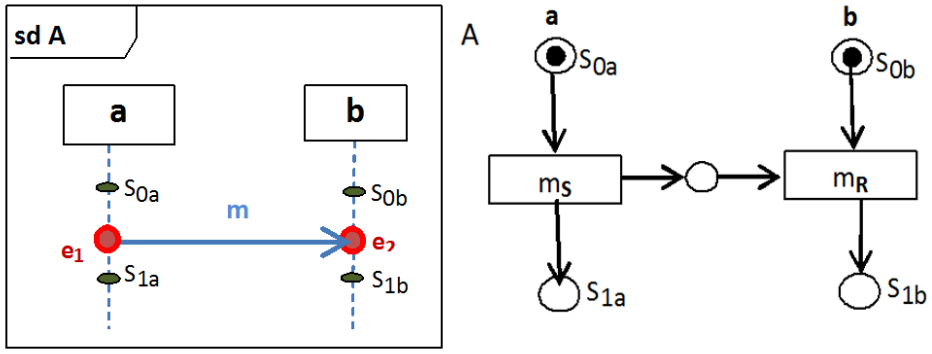
5.3 Additional Transformation Rules

UML sequence diagrams consist of two main types of interactions: synchronous and asynchronous communication (cf. Chapter 3). Synchronous communication implies that the sender and receiver complete the transition together (are blocked), whereas in the asynchronous case the sender may continue its execution after sending the message. It is common to assume one or the other forms of communication only, since they can be used to model one another. This thesis considers synchronous communication (e.g., Figure 5.2), only. For completeness we show here how to address asynchronous communication as well.

Further, the local transitions in a UML2 sequence diagrams can be categorised into four types, namely *create*, *destroy*, *lost* and *found* messages. For completeness, we illustrate the representation for the transformation of these additional types of local transitions.

5.3.1 Transformation of Asynchronous Local Transitions

When transforming an asynchronous local transition into a CPN representation, we use two separate net transitions to denote the sending event and the receiving event. Additionally, we use an intermediate place in between the sending and receiving net transitions to denote the queuing state associated with the entire asynchronous transition [Ouardani et al., 2006, dos S. Soares and Vrancken, 2008, Yang et al., 2010].



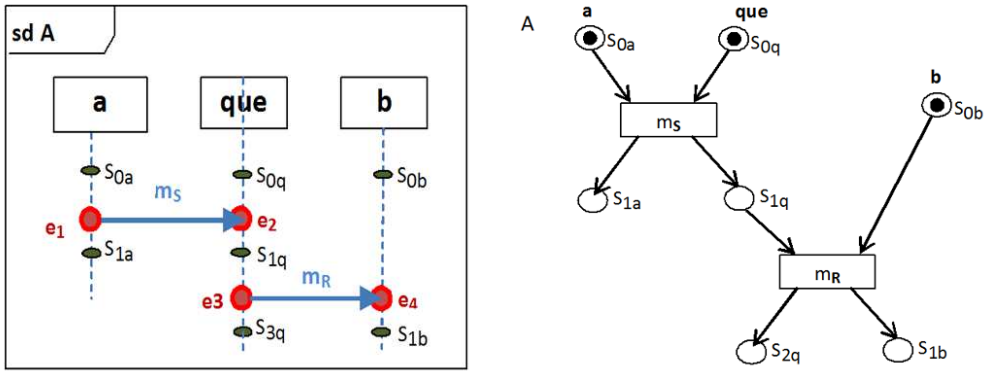
(a) (b)

Figure 5.6: A sequence diagram with an asynchronous communication (a) and the corresponding CPN (b).

Figure 5.6 shows a SD with asynchronous transition and the corresponding CPN. The CPN contains two net transitions $m_S, m_R \in T_n$ to denote as separate transitions associated with the sender and the receiver, respectively. The intermediate place has an environment colour and supports the tokens passing from m_S to m_R.

Additionally, asynchronous communication can be modelled using synchronous communication by having an intermediate role that denotes the event queue associated with an asynchronous communication. Figure 5.7 shows a SD with synchronous transitions representing an asynchronous communica-

tion, and the corresponding CPN. The instance $que \in I$ in SD_A plays the role of a mediator that received data from the sender a and forwards it to b. Further, by removing the places related to $que \in \Sigma$ in the CPN representation and having a place of $v \in \mathcal{E}$ in between m_S and m_R, we can obtain the CPN representation of an asynchronous transition (as represented in Figure 5.6 (b)).



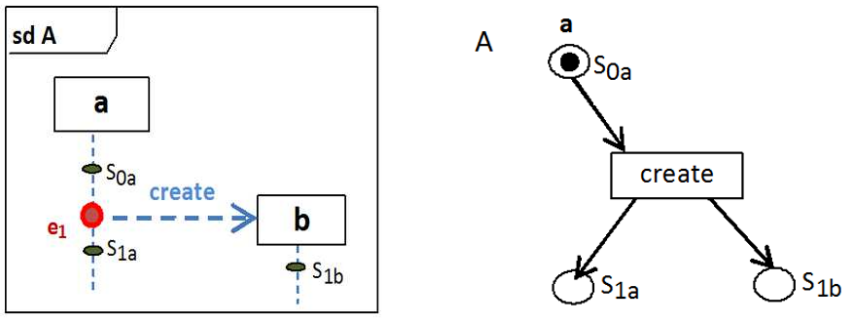
(a) (b)

Figure 5.7: A SD with synchronous communication that model asynchronous behaviour (a) and the corresponding CPN (b).

5.3.2 Transformation of Create and Destroy Transitions

In UML2 sequence diagrams new instances can be created using *create* messages. A local transition that denotes the instance creation represents using a dashed line with an open arrow pointing to the newly created instance (see Figure 5.8). By convention, the message label is typically named with some variation of *create*. The semantics of instance creation is similar to that for lifelines starting from the beginning. The only difference is that the initialisation of elements occurs when the creation message is received and there is no initial state location as in other instances. The corresponding CPN representation of a create transition can be obtained using the transformation of

a general local transition, where the newly created instance does not have a place corresponding to an initial state location.



(a) (b)

Figure 5.8: A sequence diagram with a create transition (a) and corresponding CPN (b).

The following rule defines the transformation of a *create* local transition.

Rule 5.11 *Let $t \in T$ be a create local transition with $t = (e, m, e')$, $e \in next_i(s_{0i})$, $\mu_i(m, e) = s_{1i}$ and $\mu_j(m, e') = s_{1j}$. There is a unique matching of local transitions and net transitions such that $\tau(t) = t' \in T_n^l$, and $\exists_{a_{0i}, a_{1i}, a_{1j} \in A}$: $l(t') = m$, $c(s_{0i}) = c(s_{1i}) = i$, $c(s_{1j}) = j$, and $node(a_{0i}) = (s_{0i}, t')$, $node(a_{1k}) = (t', s_{1k})$ for $k \in \{i, j\}$ and $i, j \in I^+$.*

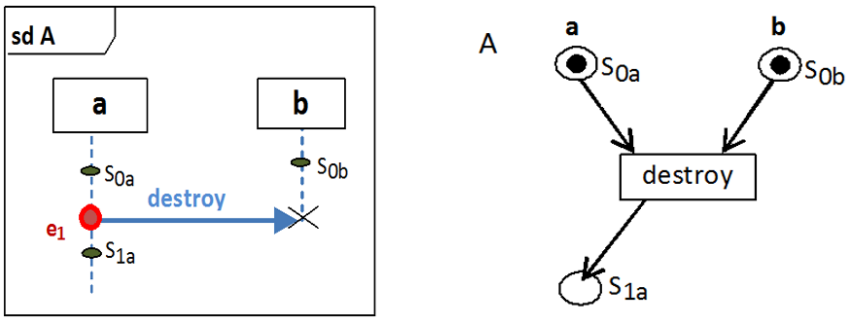
Consider the SD and the CPN model shown in Figure 5.8. The instance a in SD_A sends a *create* transition that causes the creation of an instance b. The corresponding CPN contains colours $a, b \in \Sigma$, places $S_{0a}, S_{1a}, S_{1b} \in P$, $t \in T_n$ where $l(t) = create$ and the arcs $a_{0a}, a_{1a}, a_{1b} \in A$ links the places and transitions such that $node(a_{0a}) = (S_{0a}, t), node(a_{1a}) = (t, S_{1a}), node(a_{1b}) = (t, S_{1b})$.

Moreover, UML2 sequence diagrams illustrate the destruction of an instance using a *destroy* local transition and having a large X at the end of the receiver's lifeline (see Figure 5.9). The semantics of a *destroy* transition

resulting in the deletion of the receiving instance. The corresponding CPN representation can be obtained by applying the general transformation of a local transition and by removing the last place of the instance that is deleted. Hence, the destroyed instance cannot involve in further interactions transition firing.

The following rule defines the transformation of a *destroy* local transition.

Rule 5.12 *Let* $t \in T$ *with* $t = (e, m, e^{'})$, $e \in next_i(s_{0i})$, $e^{'} \in next_j(s_{0j})$, *and* $\mu_i(m, e) = s_{1i}$. *There is a unique matching of local transitions and net transitions such that* $\tau(t) = t^{'} \in T_n^l$, *and* $\exists_{a_{0i}, a_{1i}, a_{0j} \in A}$: $l(t^{'}) = m$, $c(s_{0i}) = c(s_{1i}) = i$, $c(s_{0j}) = j$, *and* $node(a_{0k}) = (s_{0k}, t^{'})$, $node(a_{1i}) = (t^{'}, s_{1i})$ *for* $k \in \{i, j\}$ *and* $i, j \in I^{+}$.



(a) (b)

Figure 5.9: A sequence diagram with a destroy transition (a) and the corresponding CPN (b).

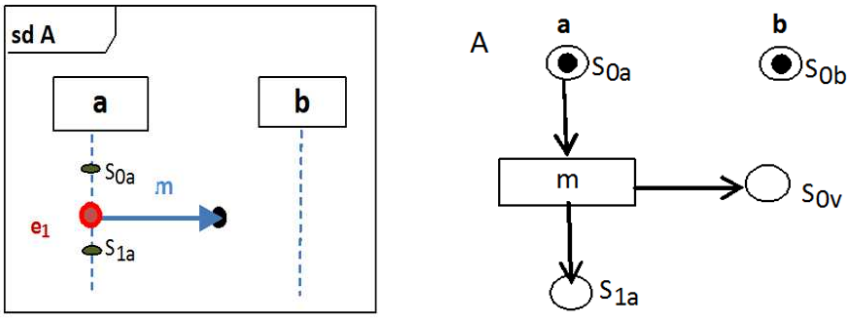
Figure 5.9 shows a SD with a *destroy* transition and the corresponding CPN. The instance a sends a *destroy* transition that causes the destruction of the instance b. The corresponding CPN model is obtained by applying Rule 5.12. Since there is no place of the colour b after the firing of the net transition *destroy*, it does not contain any more tokens of the colour b that cause further transition firings.

5.3.3 Transformation of Lost and Found Transitions

UML2 sequence diagrams may define two special types of local transitions: *lost* transitions and $found$ transitions. In a lost transition, the sending event occurrence is known, but the receiving instance is unknown, hence the reception of the message does not happen. Since the message never reached its destination, the local transition does not end on a lifeline. This situation is illustrated by pointing the arrowhead to a filled circle (see Figure 5.10). This can be considered as a local transitions connected to a gate, i.e. without an explicitly specified receiver. In the corresponding CPN representation, there are places correspond to the state locations of the sending instance and a net transition correspond to the (lost) local transition. Here, the CPN does not contain a place correspond to the receiving instance. Instead, the net transition has an arc that connects to a place of the colour environment, representing the lost token. The following rule defines the transformation of a *lost* local transition.

Rule 5.13 *Let* $t \in T$ *with a gate event such that* $t = (e, m, e^{'})$ *where* $e \in next_i(s_{0i})$, $e^{'} \in \varnothing : i \in I$ *and* $\mu_i(m, e) = s_{1i}$. *There is a unique matching of local transitions and net transitions:* $\forall_{t \in T} \exists'_{t' \in T_n^l}$, *and* $\exists_{a_{0i}, a_{1i}, a_{1v} \in A}$: $l(t^{'}) = m$, $c(s_{0i}) = c(s_{1i}) = i$, *an additional place* $s_{0v} \in P$: $c(s_{0v}) = \mathcal{E}$, *and* $node(a_{0i}) = (s_{0i}, t^{'})$ *and* $node(a_{1k}) = (t^{'}, s_{1k})$ *for* $k \in \{i, v\}$ *and* $i, v \in I^{+}$.

Figure 5.10 shows a sequence diagram with a lost local transition and the corresponding CPN. The instance a sends a local transition which gets lost during the interaction. The corresponding CPN representation contains places $S_{01}, S_{1a}, S_{0v} \in P : c(S_{01}) = c(S_{1a}) = a$ and $c(S_{0v}) = \mathcal{E}$, net transition $t \in T_n$ where $l(t) = m$, and arcs $a_{0a}, a_{1a}, a_e \in A : node(a_{0a}) = (S_{0a}, t)$, $node(a_{1a}) = (t, S_{1a})$, $node(a_e) = (t, S_{0v})$.



(a) (b)

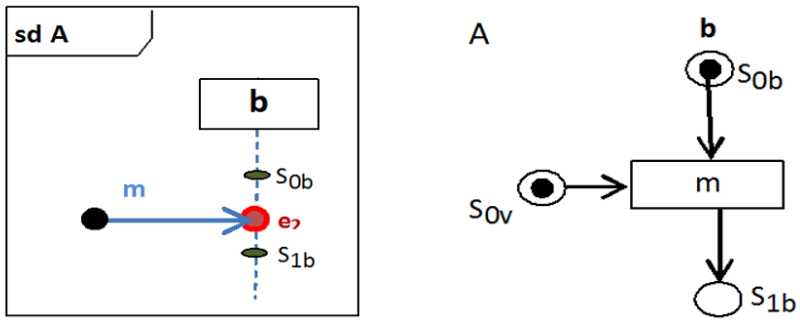
Figure 5.10: A sequence diagram with a lost transition (a) and the corresponding CPN (b).

Further, the local transition *found* is a transition where the receiving event occurrence is known, but there the sending event occurrence is unknown. Here, the origin of the transitions is considered outside the scope of the description. In a complete system design a found transition may considered as a dual of a lost transition. Thus, having matching message labels, a lost transition may be used by a found transition's receiving event resulting in a complete message transmission. In a SD, a found transition is illustrated as an arrow coming from a filled circle (see Figure 5.11). Since the sending instance is not specified, this can be considered as a transition originates from a gate. This behaviour can be transformed to a CPN representation by having the places correspond to the receiver, net transition correspond to the local transition, and additionally a place with a token of the colour environment that connects to the net transition.

The following rule defines the transformation of a *found* local transition.

Rule 5.14 *Let* $t \in T$ *with a gate event such that* $t = (e^{'}, m, e)$ *where* $e \in next_i(s_{0i})$, $e^{'} \in \varnothing : i \in I$ *and* $\mu_i(m, e) = s_{1i}$. *There is a unique matching of local transitions and net transitions:* $\forall_{t \in T} \exists^{'}_{t^{'} \in T^l_n}$, *and* $\exists_{a_{0i}, a_{1i}, a_{0v} \in A}$: $l(t^{'}) = m$,

$c(s_{0i}) = c(s_{1i}) = i$, *an additional place* $s_{0v} \in P$: $c(s_{0v}) = \mathcal{E}$, *and* $m(s_{0v}) = 1$. *Further,* $node(a_{0k}) = (s_{0k}, t^{'})$ *and* $node(a_{1i}) = (t^{'}, s_{1i})$ *for* $k \in \{i, v\}$ *and* $i, v \in I^{+}$.



(a) (b)

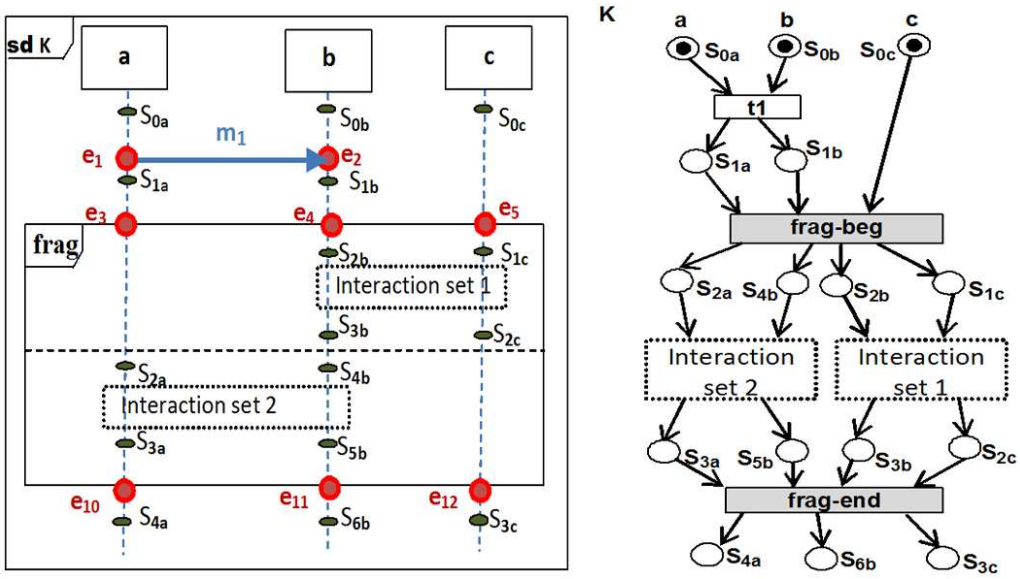
Figure 5.11: A sequence diagram with a found transition (a) and the corresponding CPN (b).

Consider the SD and the corresponding CPN shown in Figure 5.11. The instance b in SD_A receives a local transition where the sender is unknown. The corresponding CPN contains a place, S_{0v} where $c(S_{0v}) = \mathcal{E}$, and $m(S_{0v}) = 1$. The places correspond to the receiver are linked with the net transition as expected.

5.4 Transformation of Interaction Fragment Behaviour

UML2 sequence diagrams have an additional high level construct called interaction fragments that represent complex interaction behaviour within a system. Interaction fragments are denoted as frames with an operator in the left upper corner and the interaction behaviour inside. As described in Chapter 3, an interaction fragment may be built of different operands and the semantics of the diagram depends on the operator. This section defines transformations for

different behavioural types covering different application domains of sequence diagrams.



(a) (b)

Figure 5.12: A sequence diagram with an interaction fragment (a) and the corresponding CPN (b).

Entering or leaving an interaction fragment is treated here as an atomic event. Also, we assume that entering/leaving an interaction fragment is done synchronously by all the lifelines (instances) involved in the fragment. Consequently our assumed semantics has to be consistent at the CPN level. One reason for imposing synchronisation comes from the fact that local transitions may change values used in the conditions of a fragment leading to possibly unspecified behaviour.

This section starts with defining a general fragment rule, which applies to an arbitrary fragment and we then give additional rules for each fragment afterwards. Generally, net transitions in the CPN match local transitions in a SD and are labelled by the corresponding message label of the local transition.

In addition to the net transitions that match local transitions for a sequence diagram (given by Rule 5.5), a CPN may contain further net transitions associated with fragments and used to denote synchronisation of instances before and after the execution of an interaction fragment. In other words, we are assuming a SD semantics where instances can only start (or end) the behaviour described within a fragment of a SD if all instances involved in the fragment are ready (or have finished). Thus, net transitions without labels ($T_n^{\neg l}$) are a convenience introduced to impose synchronisation of object instances when entering and leaving an interaction fragment in a sequence diagram (see Figure 5.12).

Transformation of an interaction fragment from a SD to a CPN requires instance synchronisation for an arbitrary fragment and this general rule is given by the Rule 5.15.

Rule 5.15 *Let $x \in F$ be an interaction fragment in SD with $f(x) = (o, n)$, and $i \in j(x, k)$ be an arbitrary instance involved in the fragment for $1 \leq k \leq n$. Let $e_1, e_2 \in E_i$ denote the minimal and maximal event in $\overline{g(x)_i}$ respectively. Let $s_k = min(\lambda_i(x, k))$ and $s_k^{'} = max(\lambda_i(x, k))$ with $e_1 \in next_i(s)$, $next_i(e_2) = \theta_i(x) = s^{'}$ and $next_i(e_1) = \{s_1, \ldots, s_n\}$, $e_2 \in next_i(s_k^{'})$.*

CPN contains places $s, s_1, \ldots, s_n, s_1^{'}, \ldots, s_n^{'}, s^{'} \in P$, transitions $t_{o\_beg}, t_{o\_end} \in T_n^{\neg l}$, and arcs $a_{i0}, a_{i1}, \ldots, a_{in}, a_{i1}^{'}, \ldots, a_{in}^{'}, a_{i0}^{'} \in A$ such that $node(a_{i0}) = (s, t_{o\_beg})$, $node(a_{ik}) = (t_{o\_beg}, s_k)$, $node(a_{ik}^{'}) = (s_k^{'}, t_{o\_end})$, and $node(a_{i0}^{'}) = (t_{o\_end}, s^{'})$.

When transforming an interaction fragments in a SD, there exist two unlabelled net transitions in the CPN, correspond to the beginning and the end of the interaction fragment. Also there are relevant places correspond to the state locations given by $min(\lambda(x, k))$ and $max(\lambda(x, k))$, for each instance which associated with each operand. The subset of instances involved in each operand

of a fragment is given by the function $j(x, k)$ where x is the fragment identifier and k indicates the operand number.

To illustrate the transformation of a general interaction fragment, consider the interaction fragment of SD_K and the corresponding CPN in Figure 5.12. The local transitions inside each operand in the SD are represented as a block for a clear representation. These placeholders will later be substituted with the actual local transitions inside the fragment. Let x be the interaction fragment identifier. Let instance $b \in j(x,1) \cap j(x,2)$ in the SD. Note that the instance $a \notin j(x,1)$ and $c \notin j(x,2)$. For the instance b, $min(\lambda_b(x,1)) = S_{2b}$, $min(\lambda_b(x,2)) = S_{4b}$, $max(\lambda_b(x,1)) = S_{3b}$, $max(\lambda_b(x,2)) = S_{5b}$ and $\theta_b(x) = S_{6b}$.

Here, CPN_K contains two new net transitions $frag - beg$, $frag - end \in T_n^{\neg l}$ to denote the beginning and end of the interaction fragment. The net transitions inside the fragments are not explicitly specified in this example. For the colour b, the CPN contains places correspond to the state locations $S_{0b}, S_{1b}, S_{2b}, S_{3b}, S_{4b}, S_{5b}, S_{6b}$. The arcs link the places and transition as per definition, for example, $node(a_{2b}) = (S_{1b}, frag - beg)$, $node(a_{3b}) = (frag - beg, S_{2b})$, etc.

The mapping rules defined in the following section are conceptually referred to the general fragment transformation rule defined above. Thus, complex transformation rules can be constructed using basic mapping rules.

5.4.1 Transformation of Alternative Behaviour

The semantics of an *alt* interaction fragment denotes the choice of behaviour over the semantics of each operand, in which at most one of the operands will be chosen. Each operand in an alternative fragment has a condition, which is evaluated when choosing the operand to be executed. Only the interaction within the operand with a guard condition that evaluates to true is executed.

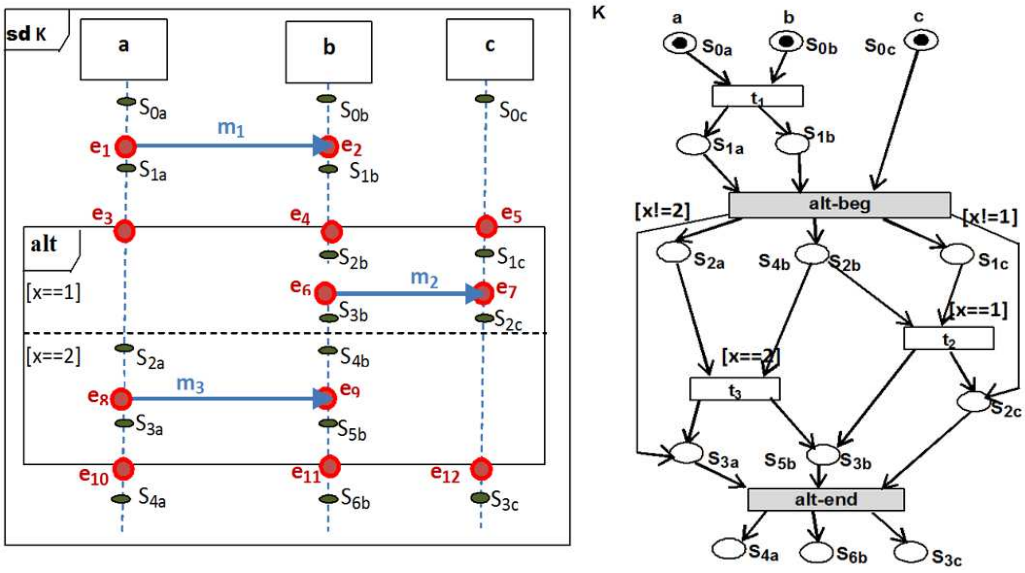
Additionally, an operand may be guarded by an *else* condition (usually the last operand in the fragment), that is the negation of the disjunction of all other guards in the enclosing interaction fragment.

The fragment synchronisation Rule 5.15 described how to obtain two net transitions for the beginning and end of an interaction fragment, and how these relate to the places derived from the state locations before the fragment, after the fragment, and within each operand.

When transforming a SD with an *alt* interaction fragment to a CPN, each operand in the SD is transformed into a sequential chain. There are many possible chains for an instance that are related to the number of operands available. All sequential chains begin in a common input place and end a common output place. This is represented in the CPN representation by introducing two net transitions, $t_{alt-beg}$ and $t_{alt-end}$ to represent the entering and leaving of the interaction fragment, respectively (see Figure 5.13). All the place and net transition chains within the net transitions $alt - beg$ and $alt - end$, describe the same sequence as in the SD.

Since only one of the operands of an *alt* fragment is executed at a time, in the CPN representation there is no need to have a set of unique places correspond to each of the first (min) state locations in each operand. This is similar to the set of last (max) state locations in each operand. Instead, we optimise them and have one place to denote the beginning of an arbitrary *alt* operand and one place to denote the end of an arbitrary *alt* operand, for each instance. For example in Figure 5.13, there is an equality between the places such that $S_{2b} = S_{4b}$ and $S_{3b} = S_{5b}$. In fact, more than an optimisation this is a necessary requirement to preserve the intended behaviour of the *alt* interaction fragment in the CPN.

Here, the condition in each operand is transformed into an expression that



(a) (b)

Figure 5.13: A sequence diagram with alternative behaviour (a) and the corresponding CPN (b).

associates with the first net transition of each chain, i.e. the first net transition within a chain is chosen based on the guard that evaluates to true. For example in Figure 5.13, $guard(t_3) = [(x == 2) == T]$ and $guard(t_1) = [(x == 1) == T]$.

Additionally, when an instance is not involved in all the operands (for example in Figure 5.13, the instance a is not involved in the first operand with the guard $[x == 1]$), and when that instance is not involved in the operand that is chosen to execute (e.g., when the first operand executes), there should be a way to pass the control flow of that instance to continue with the transitions after the net transition $t_{alt-end}$ corresponding to the end of the fragment. For that purpose, we use an arc connecting the net transition $t_{alt-beg}$ and the place corresponds to the maximum state location of that instance. Also, we use an expression which is the negation of the disjunction of all other guards

of the operands involving that instance (here, $[x! = 2]$) and associates with the newly added arc. Moreover, this can be applied for situations where the *alt* interaction fragment does not contain an *else* operand and none of the operands is evaluated to true.

This is described in the following rule.

Rule 5.16 (Alt-Rule) *Let $x \in F$ be an interaction fragment in SD with $f(x) = (alt, n)$, and $i \in j(x)$ be an arbitrary instance involved in the fragment. According to* Rule 5.15 *$s, s_1, \ldots, s_n, s_1^{'} \ldots, s_n^{'}, s^{'} \in S_{int}^{i}$ are state locations for instance i, and the associated CPN contains places $s, s_1, \ldots, s_n, s_1^{'} \ldots, s_n^{'}, s^{'} \in P$, transitions $t_{alt-beg}, t_{alt-end} \in T_n$, and arcs $a_{i0}, a_{i1}, \ldots, a_{in}, a_{i1}^{'}, \ldots, a_{in}^{'}, a_{i0}^{'} \in A$. For an alt-fragment we have the following equalities $s_1 = \cdots = s_n$, $s_1^{'} = \cdots = s_n^{'}$, $a_{i1} = \cdots = a_{in}$, and $a_{i1}^{'} = \cdots = a_{in}^{'}$. Further, if $next_i(s_k) = e_{s_k}$ and e_{s_k} is involved in a transition $t_{s_k} \in T$ as a source or target event then $guard(t_{s_k}) = [C_k == True]$ and the corresponding net transition $t_{s_k}^{'} \in T_n$ is guarded with the same expression (for all $1 \leq k \leq n$). Further, for an instance $z \in j(x) \backslash \{\forall_{k \leq n} j(x,k)\}$, we use an additional arc $a_z^{''} \in A$, such that $node(a_z^{''}) = (t_{alt-beg}, s_k^{'})$, where $s_k^{'} = max_z(\lambda_i(x,k))$, and guarded with the negation of the disjunction of all other guards such that $guard(a_z^{''}) = [C_k! = True]$.*

By applying Rule 5.16 to an *alt* interaction fragment, a CPN model can be obtained that describes the choice of behaviour in the same way. Consider the SD with an *alt* interaction fragment with two operands and the corresponding CPN shown in Figure 5.13. SD_K contains three object instances $a, b, c \in I$. The interactions start with a local transition t_1 followed by an interaction fragment x such that $f(x) = (alt, 2)$. The state locations in each operand for the instance $b \in j(x,1) \cap j(x,2)$ are given by $\lambda_b(x,1) = \{S_{2b}, S_{3b}\}$, $\lambda_b(x,2) = \{S_{4b}, S_{5b}\}$.

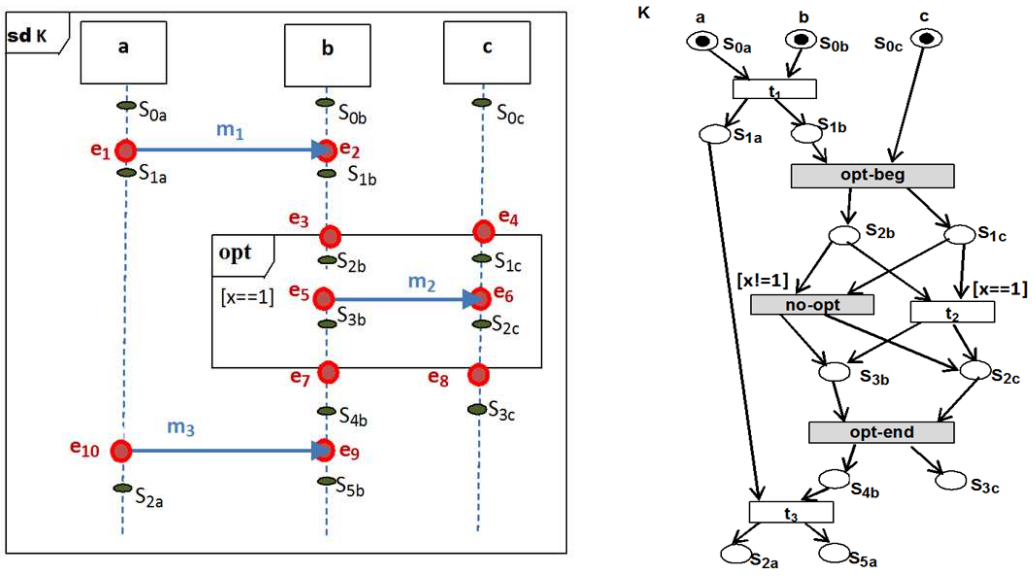
Applying Rule 5.16 the corresponding CPN model contains two new net transitions, $alt - beg, alt - end \in T_n^{\neg l}$, correspond to the begin and end of the interaction fragment, respectively. Also, for the colour b, CPN contain places: $S_{2b}, S_{3b}, S_{4b}, S_{5b} \in P$ that satisfy the equalities $S_{2b} = S_{4b}$ and $S_{3b} = S_{5b}$. Similar equality applies for the arcs that are linked with these plaes. Further, the net transitions $t_2, t_3 \in T_n$, correspond to the local transitions in the fragment are labelled with the message labels such that $l(t_2) = m_2$ and $l(t_3) = m_3$. These net transitions correspond to the first local transition in each operand, and are guarded such that $guard(t_2) = [(x == 1) = True]$ and $guard(t_3) = [(x == 2) = True]$.

5.4.2 Transformation of Optional Behaviour

The *opt* interaction fragment denotes a choice of behaviour where either the operand happens or not. This optional behaviour can be considered as an *alt* interaction fragment with only one operand. The interactions within the operand execute, if the guard condition is evaluated to true. If the guard condition is evaluated to false, then the interactions within the *opt* operand are ignored, the fragment discarded, and the remainder of the interaction in the SD are continued with execution.

When transforming a SD with an *opt* interaction fragment, the corresponding CPN representation contains two net transitions, $opt - beg$ and $opt - end$ (according to Rule 5.15) to synchronise the behaviour at the beginning and end of the interaction fragment, respectively. All the places and net transitions chains within the net transitions $opt - beg$ and $opt - end$ describe the same sequence as in the SD. The guard condition associates with the *opt* fragment is associated with the first net transition after $opt - beg$. When the guard is evaluated to true, the behaviour of the CPN is the same as the *alt* interaction

fragment transformation and no further restrictions are necessary.



(a) (b)

Figure 5.14: A sequence diagram with optional behaviour (a) and the corresponding CPN (b).

In this case, when deriving the underlying CPN, as shown in Figure 5.14(b), a new net transition, $no - opt$, is defined. The net transition $no - opt$ is associated with a guard condition that is the negation of the disjunction of the condition in the enclosing *opt* interaction fragment. This new net transition is linked with the places that correspond to the minimum and maximum state locations within the fragment, of an instance.

Rule 5.15 described how to obtain the two net-transitions for the beginning and end of an interaction fragment, and their connection to the places derived from the state-locations before and after the fragment, and within each operand. The following rule describes the transformation of an *opt* interaction fragment in a SD to a CPN.

Rule 5.17 (Opt-Rule) *Let $x \in F$ be an interaction fragment in SD with $f(x) = (opt, 1)$, and $i \in j(x, 1)$ be an arbitrary instance involved in the fragment. According to Rule 5.15 $s, s_1, s_1^{'}, s^{'} \in S_{int}^{i}$ are state locations for instance i, and the associated CPN contains places $s, s_1, s_1^{'}, s^{'} \in P$, transitions $t_{opt-beg}$, $t_{opt-end} \in T_n^{\neg l}$, and arcs $a_{i0}, a_{i1}, a_{i1}^{'}, a_{i0}^{'} \in A$.*

For an opt fragment we have additionally a new net-transition $t_{no-opt} \in T_n^{\neg l}$, and arcs $a_i^{''}, a_i^{'''} \in A$, such that $node(a_i^{''}) = (s_1, t_{no-opt})$, and $node(a_i^{'''}) = (t_{no-opt}, s_1^{'})$. Further, the opt expression given by $guard(x) = [C == True]$ is associated with the net transition $t \in T_n$, which corresponds to the first local transition within the opt fragment: $guard(t) = [C == True]$. The new net-transition t_{no-opt} is guarded with the negated disjunction of the expression: $guard(t_{no-opt}) = [C! = True]$.

In order to explain Rule 5.17, consider the *opt* interaction fragment in the SD shown in Figure 5.14. Let x be the identifier of the *opt* fragment and consider the instance b involved in the fragment: $b \in j(x, 1)$. The state locations in the operand for the instance b are given by $\lambda_b(x, 1) = \{S_{2b}, S_{3b}\}$.

Applying Rule 5.17, we derive the CPN with corresponding places $S_{2b}, S_{3b} \in P$ of the colour b. Also, we obtain the new net-transition $t_{no-opt} \in T_n^{\neg l}$ and the arcs $a_1, a_2 \in A$ link the places with the new net-transition, in such a way that, $node(a_1) = (S_{2b}, t_{no-opt})$, $node(a_2) = (t_{no-opt}, S_{3b})$. Further, the condition associated with the fragment x is guarded as expected: $guard(t_2) = [C == True]$ and $guard(t_{no-opt}) = [C! = True]$, which is the negated disjunction of the *opt* guard. So that, when the guard evaluates to true the net transition t_2 fires. Conversely, when the guard evaluates to false, the execution flow reaches to the net transition $t_{opt-end}$ via the net transition t_{no-opt}.

5.4.3 Transformation of Iterative Behaviour

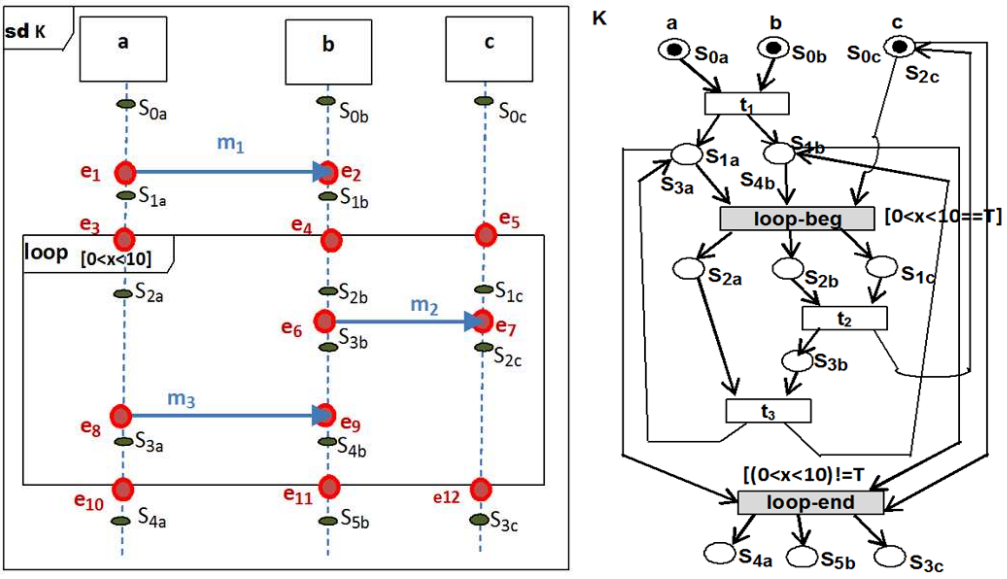
The *loop* interaction fragment indicates iterative behaviour. The behaviour specified inside the loop operand executes repeatedly until the loop guard (if available) evaluates to false. This iterative behaviour is controlled by a guard or by an expression with minimum and maximum number of iterations. When the condition is given by two integer parameters $0 \leq min \leq max \leq \infty$, the interaction within the fragment is processed at least min times and at most max times. The interactions within the fragment can be bypassed if either the condition of the loop is not satisfied, or min is zero, in which case the loop is not executed.

When transforming a *loop* fragment in a SD to an equivalent behaviour in the CPN, the corresponding CPN model contains net transitions $loop - beg$ and $loop - end$ to denote the beginning and the end of the loop fragment and related places inside the operand as given by Rule 5.15.

Consider the CPN shown in Figure 5.15 that guarantee the repetitive behaviour. To adapt Rule 5.15 for a *loop* fragment we impose an equality for the places correspond to the state location (S_{1a}) before the beginning of the *loop* fragment and last state location (S_{3a}) inside the operand, for each instance involved in the fragment. Also, there is a corresponding expression in the CPN for the guard inside the loop fragment and associate with the beginning of the *loop* interaction fragment (evaluating to true), as well as to the net transition $loop - end$ (evaluating to false).

Here, we associate this expression with the unlabelled net transition $loop - beg$ that corresponds to the beginning of the fragment. Further, the net-transition $loop - end$ that corresponds to the end of the interaction fragment is associated with the negation of the condition, that is, the loop guard evaluated to false. These expressions are tested on each time the loop iterates and the

firing net transition is selected based on the satisfied condition. Additionally, to control the number of iterations, a variable is used as a counter that increments in each execution of the net transition $loop - beg$. This variable corresponds to a variable considered in the expression of the *loop* fragment and initialised with the minimum value given by the expression. The net transitions within $loop - beg$ and $loop - end$ fires only if the guard condition associated with the $loop - beg$ evaluated to true. The net transition $loop - end$ is enabled when the negation of the guard condition evaluated to true, i.e. when the guard condition evaluates to false.



(a) (b)

Figure 5.15: A sequence diagram with iterative behaviour (a) and the corresponding CPN (b).

The transformation of a *loop* fragment to a CPN is defined in Rule 5.18.

Rule 5.18 (loop-Rule) *Let $x \in F$ be an interaction fragment in SD with $f(x) = (loop, n)$, and $i \in j(x, 1)$ be an arbitrary instance involved in the fragment. According to Rule 5.15 $s, s_1, s_1^{'}, s^{'} \in S_{int}^{i}$ are state locations for instance i, and the associated CPN contains places $s, s_1, s_1^{'}, s^{'} \in P$, transitions $t_{loop-beg}, t_{loop-end} \in T_n^{\neg l}$, and arcs $a_{i0}, a_{i1}, a_{i1}^{'}, a_{i0}^{'} \in A$.*

For a loop fragment we have additionally the equality $s = s_1^{'}$. Similarly, the net transition $t_{loop-end}$ is guarded with the negated disjunction of the loop guard expression: $guard(t_{loop-end}) = [C! = True]$. Furthermore, let the loop condition be $C = [min \leq v \leq max]$: with a counter variable $v \in X$, then for each $guard(t_{loop-beg}) = [C == T]$, $v = v + 1$.

Figure 5.15(a) shows a sequence diagram with a *loop* fragment and Figure 5.15(b) shows the corresponding CPN model. SD_K initiate with a local transition $t_1 = (e_1, m_1, e2)$ followed by a *loop* interaction fragment id : $guard(id) = [C == True]$, where $C = [0 \leq x \leq 10]$, $C \in Exp$ and $x \in X$. Consider the instance $b \in j(id)$ that involves in the fragment. The state locations in the operand for the instance b are given by $\lambda_b(id, 1) = \{S_{2b}, S_{3b}, S_{4b}\}$.

The corresponding CPN, which is derived by applying Rule 5.18, has places $S_{2b}, S_{3b}, S_{4b} \in P$ of the colour b whereby $S_{1b} = S_{4b}$. The net-transition $loop - beg \in T_n^{\neg l}$, which corresponds to the beginning of the *loop* fragment, is guarded by $gaurd(loop - beg) = [C == True]$; whereas the net-transition $loop{-}end \in T_n^{\neg l}$, is guarded by $guard(loop{-}end) = [C! = True]$. Furthermore, the corresponding variable x increments each time when $gaurd(loop - beg) = [C == True]$: $x = x + 1$. Hence, the number of possible iterations depends on the condition associated with the *loop* operand.

5.4.4 Transformation of Break Behaviour

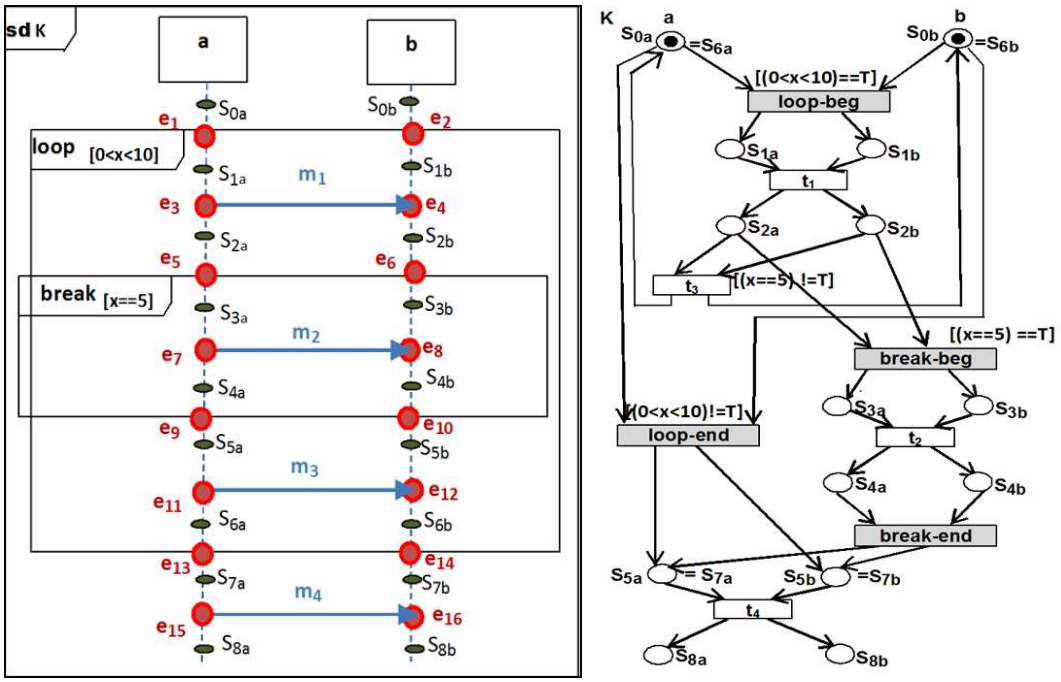
The *break* interaction fragment represents a breaking situation usually within a loop interaction fragment. The operand in the *break* fragment is associated with a guard expression and when the condition is evaluated to true, the interaction within the *break* operand happens and it ignores the remainder of the enclosing interaction fragment (*loop*) and continues with the interactions after the *loop* fragment. When the guard expression is evaluated to false, the *break* operand is ignored and the rest of the scenario within the *loop* interaction fragment happens. The behaviour is equivalent to that of an alternative fragment with the contents of the *break* fragment as one operand and all remaining elements of the diagram as an *else* branch.

The transformation of a *break* fragment nested within a *loop* fragment, to a CPN model can be considered in two ways: (1.) when the *loop* fragment contains interaction after the *break* fragment (Figure 5.16(b)) and (2.) when the *loop* fragment does not contain any interaction after the *break* fragment Figure 5.17(b). In both cases the CPN representation contains $break - beg$ and $break - end$ net transitions to indicate the beginning and the end of the *break* interaction fragment.

Here, the *break* guard condition is associated with the net transition $break - beg$. Additionally we impose an equality between the places correspond to the state locations after the *break* fragment and after the *loop* fragment. Thus, both net transitions $break - end$ and $loop - end$ connect to the same place. Hence, when the *break* fragment executes, it terminates the *loop* fragment without firing the rest of the interactions within the loop.

In the former case (1), we associate the negation of the disjunction of the condition in the enclosing *break* interaction fragment with the net transition that corresponds to the first local transition after the *break* fragment and

within the *loop* fragment (here, the net transition t_3). Thus based on the condition that evaluates to true, the firing net transitions can be selected; i.e. the interaction within the *break* fragment or the remaining interaction within the *loop* fragment.



(a) (b)

Figure 5.16: A sequence diagram with break behaviour: Case II (a) and the corresponding CPN (b).

In the latter case (2), when the SD does not contain any interactions after the *break* fragment within the *loop* fragment, a new net transition named $no-break$ is added. The net transition $t_{no-break}$ is associated with a guard condition that is the negation of the disjunction of the condition in the enclosing *break* interaction fragment. This new net transition is connected from the places that correspond to the state locations before the *break* fragment and connected to the places that correspond to the state locations before the *loop* fragment for each instance involved in the fragment. Hence, when the *break* condition is

not satisfied the execution control returns to the *loop* fragment.

We define this transformation as follows reusing Rule 5.18.

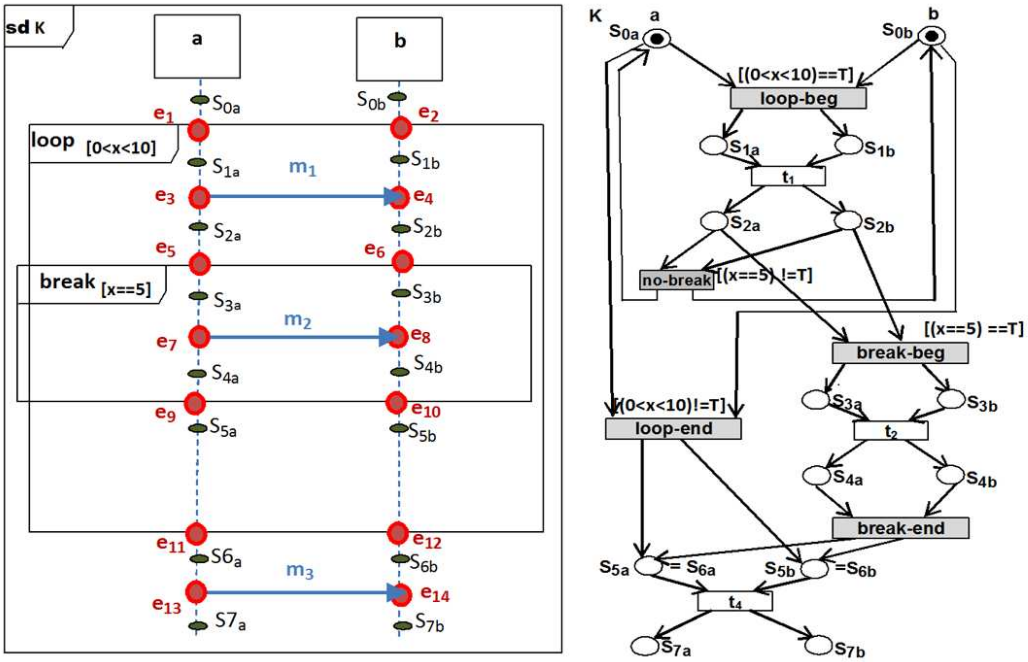
Rule 5.19 (Break-Rule) *Let $x, y \in F$ be interaction fragments in SD with $f(y) = (loop, 1)$, $f(x) = (break, 1)$, such that $h(y, 1) = x$ and $i \in j(y) \cap j(x)$ be an arbitrary instance involved in the fragment. Applying Rule 5.15 to the fragment x, $s, s_1, s_1^{'}, s^{'} \in S_{int}^{i}$ are state locations for instance i, and the corresponding CPN contains places $s, s_1, s_1^{'}, s^{'} \in P$, and transitions $t_{break-beg}, t_{break-end} \in T_n^{\neg l}$.*

Let $\theta_i(y) = s^{''}, \theta_i(x) = s^{'} \in S_{int}^{i}$ be the state locations after the loop and break fragments for instance i, respectively, and the associated CPN contains places $s^{''}, s^{'} \in P$. For the condition associated with the break fragment, $guard(x) = [C == True]$, the corresponding net transition $t_{break-beg}$ is guarded with the same expression.

Case I: When there are interactions after the break and within the loop fragment: Let $t = (e, m, e^{'}) \in T$ be the first local transition in the loop fragment after the break fragment: $(next(s^{'}) = e) \cup (next(s^{'}) = e^{'})$. The corresponding net transition $t^{'} \in T_n$ is guarded with the negated disjunction of the break guard expression: $guard(t^{'}) = [C! = True]$ and connected from the place s : $node(a_{0i}) = (s, t^{'})$.

Case II: When there are no interactions after the break and within the loop fragment: Applying Rule 5.15 to the fragment y, let $s_{loop} \in S_{int}^{i}$ be the state location before the beginning of the loop for instance i, and the corresponding CPN contains place $s_{loop} \in P$. In this case, we have additionally a new net-transition $t_{no-break} \in T_n^{\neg l}$, and arcs $a_i^{''}, a_i^{'''} \in A$, such that $node(a_i^{''}) = (s, t_{no-break})$, and $node(a_i^{'''}) = (t_{no-break}, s_{loop})$. The new net-transition $t_{no-break}$ is guarded with the negated disjunction of the expression: $guard(t_{no-break}) = [C! = True]$.

Additionally, for a break fragment we have additionally the equality between the places: $s^{'} = s^{''}$.



(a) (b)

Figure 5.17: A sequence diagram with break behaviour: Case I (a) and the corresponding CPN (b).

To illustrate the transformation of a *break* fragment nested in a *loop* fragment, consider the SD and the corresponding CPN shown in Figure 5.16. Let id_l and id_b be the fragment identifiers of the *loop* and the *break* interaction fragments, respectively. Here, $a \in j(id_l) \cap j(id_b)$ is an instance involved in both fragments. The state locations in the *break* operand for the instance a are given by $\lambda_a(id_b, 1) = \{S_{3a}, S_{4a}\}$. The state location before the beginning and after the end of the *break* fragment are such that $next_a(S_{2a}) = e_5$ and $\theta_a(id_b) = next_a(e_9) = S_{5a}$ respectively. Similarly, for the *loop* fragment, $next_a(S_{0a}) = e_1$ and $\theta_a(id_l) = next_a(e_{13}) = S_{7a}$.

CPN_K is obtained by applying the transformation given by Rule 5.19 and the corresponding CPN contain places $S_{5a}, S_{7a} \in P$ of colour a, where by $S_{5a} = S_{7a}$. The *break* guard condition $C = [x == 5]$ is associated with the net transition correspond to the beginning of the *break* fragment : $guard(break - beg) = [C == True]$. Further, the negated disjunction of the condition is associated with the net transition correspond to the first local transition after the *break* within the *loop* fragment such that $guard(t_3) = [C! = True]$.

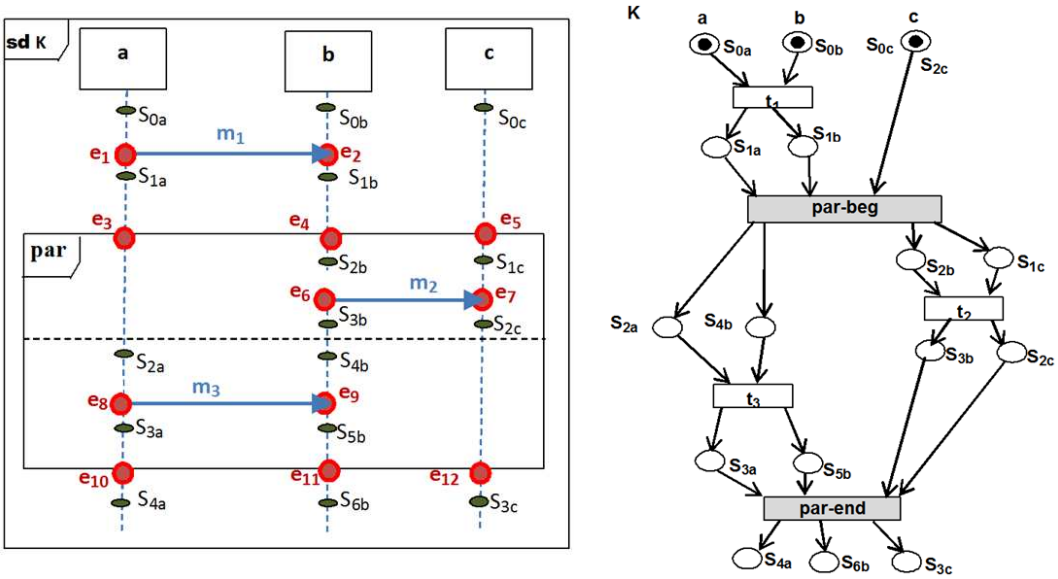
Consider the transformation of case II, where the *loop* fragment does not contain any interactions after the *break* fragment. Figure 5.17 shows a SD and the corresponding CPN representation with this behaviour. The CPN contains an additional net transition $t_{no-break} \in T_n^{\neg l}$ and the arcs $a_1, a_2 \in A$ link the places with the new net-transition, in such a way that, $node(a_1) = (S_{2a}, t_{no-break})$, $node(a_2) = (t_{no-break}, S_{0a})$. Further, the negated disjunction of the *break* guard condition is associated with this net transition as expected: $guard(t_{no-break}) = [C! = True]$. Thus when the *break* guard condition is not satisfied the net transition $t_{no-break}$ fires and the execution flow goes back to the interaction within the loop.

5.4.5 Transformation of Parallel Behaviour

The operator *par* represents a parallel execution of the behaviours of the operands. The occurrences of the different operands can be interleaved in any way, while the execution order of local transitions inside each operand is preserved. This operator has a natural representation with the CPN model, which supports the description of concurrency and parallelism.

According to Rule 5.15 , the CPN model corresponding to the behaviour of the *par* interaction fragment has two additional transitions to synchronise the control at the beginning and the end of a *par* interaction fragment (Fig-

ure 5.18). The net transition $t_{par-beg}$ creates branches for each operand, passing a token into the linked places in each operand (fork operation). This supports the interleaving between the transitions of each branch. The net transition $t_{par-end}$ has to wait for the execution of all branches to complete as it can only fire when all it's input places have tokens available (join operation).



(a) (b)

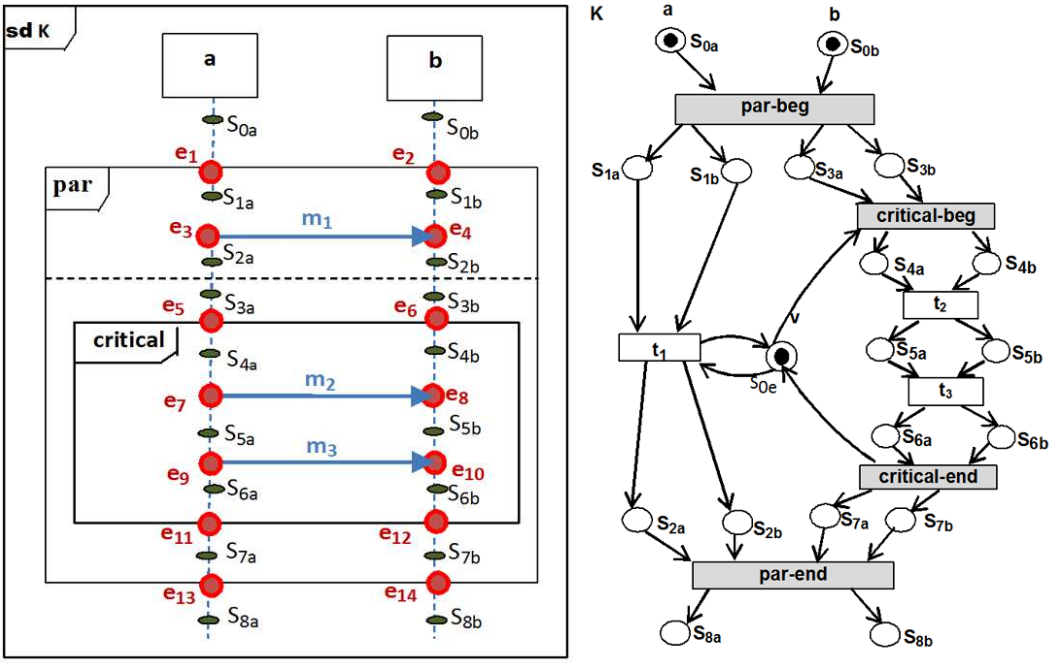
Figure 5.18: A sequence diagram with parallel behaviour (a) and the corresponding CPN (b).

When deriving a CPN, we can obtain a model that describes this concurrent behaviour in the same way. In fact, previously described Rule 5.15 is all we need to describe a *par* fragment and no further restrictions are necessary. This is illustrated by the example in Figure 5.18. SD_K consists of a parallel fragment x with two operands. The state locations of the instance b in each operand are given by $\lambda_b(x,1) = \{S_{2b}, S_{3b}\}$, $\lambda_b(x,2) = \{S_{4b}, S_{5b}\}$. Applying Rule 5.15, the CPN has places $S_{2b}, S_{3b}, S_{4b}, S_{5b} \in P$ of colour b. The net transitions $t_{par-beg}, t_{par-end}$ synchronise the instances involved in the fragment.

It is straightforward to apply this transformation to more than two parallel operands by just adding more output places to the fork net transition and, similarly, more input places to the join net transition.

5.4.6 Transformation of Critical Behaviour

The behaviour of a *critical* interaction fragment indicates that the given interaction within the fragment are treated as an atomic block. *Critical* fragments are typically used inside a *par* interaction fragment to ensure that a group of interactions cannot be separated or interleaved with other transitions. In other words, the local transitions given within a *critical* interaction fragment must execute atomically without interruption.



(a) (b)

Figure 5.19: A sequence diagram with critical behaviour (a) and the corresponding CPN (b).

In order to represent the behaviour of a *critical* fragment nested in a *par*

interaction fragment in a CPN model, we establish means to exclude the execution of interleaving transitions in other parallel operands, when the transitions within the *critical* behaviour are firing. This is achieved by an additional place with a token of the colour *environment* for each net transition that belongs to the *par* fragment excluding the *critical* fragment, and having in/out arcs between the net transition and the new place. Thus, each concurrent transition is checked for an existing environment token each time it fires.

Further, we link each of these places with the net transitions that correspond to the beginning and the end of the *critical* fragments using out arc and in arc, respectively (Figure 5.19). Hence, the critical section would collect all such tokens on entering the crucial behaviour, thus stopping all concurrent net transition executions correspond to the other operands. Since the tokens are put back to the relevant environment places after the critical section is completed, the concurrent executions of other net transitions may continue. Without loss of generality and due to enhanced readability, we introduced these concepts only for one net transition in another operand.

The following rule describes the transformation of *critical* behaviour nested within a *par* fragment.

Rule 5.20 (Critical-Rule) *Let $x, y \in F$ be interaction fragments in SD with $f(y) = (par, n)$, $f(x) = (critical, 1)$, such that $h(y, 1) = x$ and $i \in j(y) \cap j(x)$ be an arbitrary instance involved in the fragment. Applying Rule 5.15 to the fragment x, the corresponding CPN contains transitions $t_{critical-beg}, t_{critical-end} \in T_n^{\neg l}$. Let $t^{'} \in T_n$ be an arbitrary net transition that correspond to a local transition in SD: $t = (e, m, e^{'}) \in T$, where $e, e^{'} \in g(y)$ and $e, e^{'} \notin g(x)$. Additionally, $\forall t^{'}$ we have a new place $p \in P$ of the environment colour: $c(p) = \mathcal{E}$, $m(p) = 1$ and arcs $a_1, a_2, a_3, a_4 \in A$: $node(a_1) = (p, t_{critical-beg})$, $node(a_2) = (t_{critical-end}, p)$, $node(a_3) = (p, t^{'})$ and $node(a_4) = (t^{'}, p)$.*

Consider Figure 5.19(a) that shows a *critical* fragment nested in a *par* fragment. The local transitions enclosed within the *critical* interaction fragment (here, m_2 and m_3) must execute as a one unit and cannot be interleaved with other local transitions (here m_1). In other words, $m_1 \cdot m_2 \cdot m_3$ and $m_2 \cdot m_3 \cdot m_1$ are valid traces, whereas $m_2 \cdot m_1 \cdot m_3$ is invalid. The corresponding CPN shown in Figure 5.19(b) can be obtained by applying Rule 5.20.

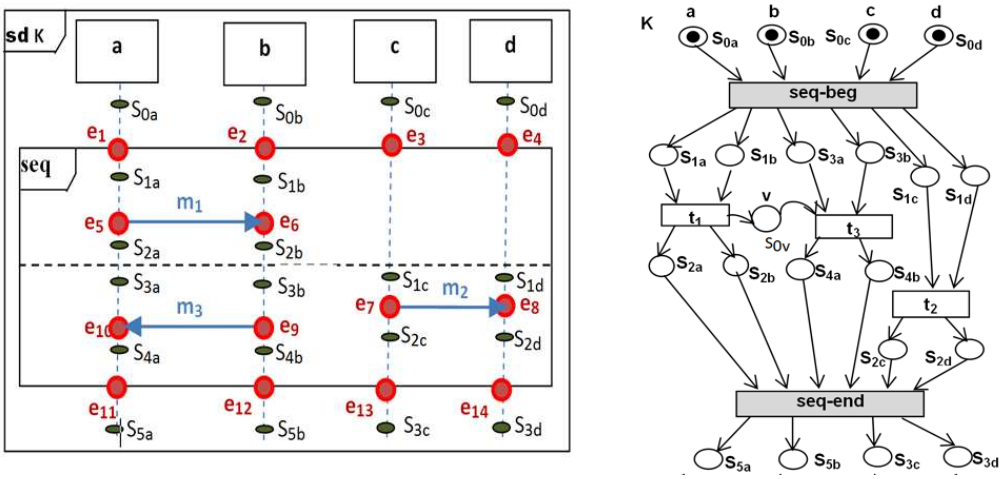
CPN_K contains an additional place S_{0e} for the environment such that $c(S_{0e}) = v$, $v \in \mathcal{E}$ and $m(S_{0e}) = 1$. Let $a_1, a_2, a_3, a_4 \in A$ be the additional arcs that link the environment place with the net transitions such that $node(a_1) = (S_{0e}, t_{critical-beg})$, $node(a_2) = (t_{critical-end}, S_{0e})$, $node(a_3) = (S_{0e}, t^{'})$ and $node(a_4) = (t^{'}, S_{0e})$. As can be seen from this example, the token in S_{0e} is consumed when entering the critical region and only released once the critical region is completed. The execution of t_1 requires a token to be available in S_{0e}.

5.4.7 Transformation of Sequence Behaviour

The interaction fragment *seq* represents a weak sequencing of the behaviour of the operands. The local transitions within each operand, provided they share a lifeline, are executed in sequence as expected given the order in which they appear. Similarly, the local transitions from different operands that concern the same instance (lifeline) are executed in the order shown. That is, an occurrence of the first operand comes before one in the second operand, and so on. When the involving instances are mutually exclusive, the local transitions may execute in any order. Thus, weak sequencing defines local causality inside and between operands of an interaction fragment, when they share same instances. This behaviour reduces to *strict* behaviour when the operands work on only one participant. Moreover, when the local transitions in operands

are involved with disjoint set of instances, this behaviour reduces to parallel behaviour. [OMG, 2011a].

In order to preserve the behaviour of a *seq* fragment using a CPN model, we have to establish means to prohibit parallel execution of net transitions in different operands that share the same instance and preserve the global ordering of the net transition firing along a lifeline. This is achieved by having an additional place of the environment colour, in between the last transition of an operand and the first transition of the next operand, only if the same instance involved in the operands. Here, the transitions in the next operand that involve with places of same colour, get tokens to fire only after the transitions within the current operand are completed. This preserves the *seq* ordering along a given colour while stopping parallel executions.



(a) (b)

Figure 5.20: Sequence diagram with sequential behaviour and corresponding CPN.

The following rule derives a CPN, which describes the weak sequencing behaviour in the same way.

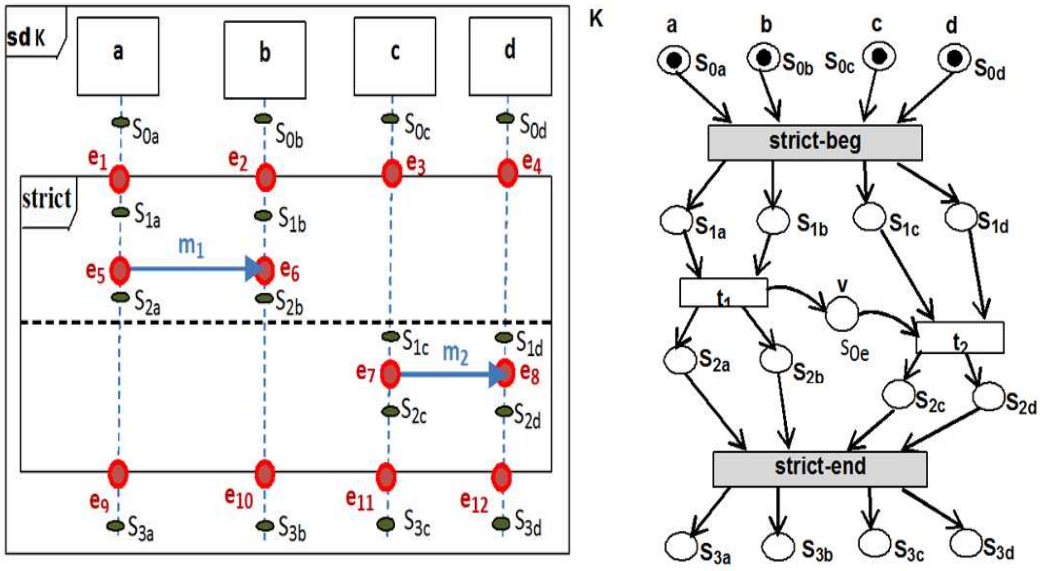
Rule 5.21 (Seq-Rule) *Let $x \in F$ be interaction fragments in SD with $f(x) = (seq, n)$, and $i \in j(x,k) \cap j(x,(k+1))$ for $k \in \mathbb{N}$ and $k < n$, be an arbitrary instance involved in the operand k and $(k+1)$. Applying Rule 5.15 to the fragment x, the corresponding CPN contains places, transitions and arcs as expected.*

Let $t_1 = (e_1, m_1, e_2), t_2 = (e_3, m_2, e_4) \in T$ be the maximum and minimum local transitions associate with the shared instance i involved in two operands, such that $e_1, e_2, e_3, e_4 \in E$, where $e_1 \cup e_2 \in max(g_i(x,k))$, and $e_3 \cup e_4 \in min(g_i(x,(k+1)))$, and the corresponding net transitions be $t_1^{'}, t_2^{'} \in T_n$. Additionally, $\forall t_1^{'}$ we have a new place $p \in P$ of the environment colour: $c(p) = \mathcal{E}$ and arcs $a_1, a_2 \in A$: $node(a_1) = (t_1^{'}, p)$ and $node(a_2) = (p, t_2^{'})$.

Consider the example shown in Figure 5.20. SD_K contains a interaction fragment $f(x) = (seq, 2)$ with local transitions $t_1 = (e_5, m_1, e_6)$, $t_3 = (e_9, m_3, e_{10})$ in two operands and sharing the same set of instances a, b. The local transition $t_2 = (e_7, m_2, e_8)$ involving instances c, d are mutually exclusive from the other transitions. Here t_1 is the last transition in the operand 1 : $e_5, e_6 \in max(g(x,1))$ and t_3 is the first transition in the operand 2 w.r.t. the shared instances a and b : $e_9, e_{10} \in min(g(x,2))$. By applying Rule 5.21, we derive the CPN in Figure 5.20(b), with net-transitions $strict - beg, strict - end \in T_n^{\neg l}$, to synchronise the behaviour given by the weak sequencing and the corresponding net transitions $t_1, t_2, t_3 \in T_n$. Further, there is a new place, $S_{0v} \in P$ of the colour environment $v \in \mathcal{E}$ in between t_1 and t_3 and arcs $a_1, a_2 \in A$ link the place with the net transitions such that $node(a_1) = (t_1, S_{0e})$ and $node(a_2) = (S_{0e}, t_3)$. Here, the net transitions t_2 may be interleaved with others in any way.

5.4.8 Transformation of Strict Behaviour

The *seq* interaction fragment (or the default interaction behaviour) only imposes an execution order on transitions that have a shared instance and completely independent transitions are interleaved in any way. A more strict order of execution can be imposed by a *strict* interaction fragment, which applies to all instances involved in the interaction fragment. Each operand in a strict fragment is executed before the next operand, and so on, and imposes a strict execution order between the behaviour of operands.



(a) (b)

Figure 5.21: A sequence diagram with strict behaviour (a) and the corresponding CPN (b).

In order to preserve the behaviour of a *strict* fragment using a CPN model, we have to establish means to prohibit execution of net transitions that are involved in mutually exclusive instance colours and preserve the total ordering of the net transition firing. This is achieved by having an additional place of the environment colour in between the net transitions correspond to the last

transition of an operand and the first transition of the next operand within the strict behaviour. So that the net transitions correspond to the local transitions in the next operand will get tokens to fire only after the net transitions correspond to the local transitions within the current operand are completed. This preserves the strict ordering, while stopping parallel executions of the net transitions that associate with mutual exclusive colours.

The following rule derives a CPN, which describes the strict sequencing behaviour in the same way.

Rule 5.22 (Strict-Rule) *Let $x \in F$ be interaction fragments in SD with $f(x) = (strict, n)$ for $n \in \mathbb{N}$. Applying Rule 5.15 to the fragment x, the corresponding CPN contains places, transitions and arcs as expected. Also, Applying Rule 5.21 the corresponding CPN gives the seq behaviour.*

Additionally, let $a, b \notin j(x,k) \cap j(x,(k+1))$ for $k < n$, be mutually exclusive arbitrary instances involved in the operand k and $(k+1)$. Let $t_1 = (e_1, m_1, e_2), t_2 = (e_3, m_2, e_4) \in T$ be the maximum and minimum local transitions associate with the mutual exclusive instances a, b involved in two operands, such that $e_1, e_2, e_3, e_4 \in E$, where $e_1 \cup e_2 \in max(g_a(x,k))$, and $e_3 \cup e_4 \in min(g_b(x,(k+1)))$, and the corresponding net transitions be $t_1^{'}, t_2^{'} \in T_n$.

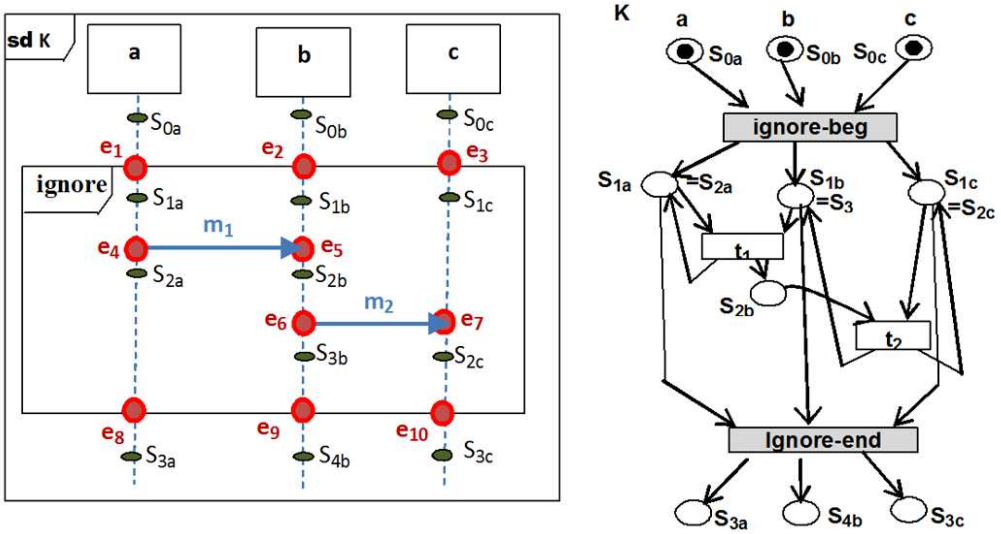
Here, $\forall t_1^{'}$ we have a new place $p \in P$ of the environment colour: $c(p) = \mathcal{E}$ and arcs $a_1, a_2 \in A$: $node(a_1) = (t_1^{'}, p)$ and $node(a_2) = (p, t_2^{'})$.

This is explained by the example shown in Figure 5.21. SD_K contains a *strict* interaction fragment with two local transitions $t_1 = (e_5, m_1, e_6)$ and $t_2 = (e_7, m_2, e_8)$ where the involving instances a, b and c, d are mutually exclusive. By applying Rule 5.22, we derive the CPN in Figure 5.21(b), with net-transitions $strict - beg, strict - end \in T_n^{\neg l}$, to synchronise the behaviour given by the strict sequencing and the net transitions $t_1, t_2 \in T_n$ correspond to the local transitions within the *strict* fragment. Further, there is a new

place, $S_{0e} \in P$ of the colour environment $v \in \mathcal{E}$ in between t_1 and t_2 and arcs $a_1, a_2 \in A$ link the place with the net transitions such that $node(a_1) = (t_1, S_{0e})$ and $node(a_2) = (S_{0e}, t_2)$.

5.4.9 Transformation of Ignore Behaviour

The *ignore* interaction fragment specifies interactions that are intentionally disregarded from the present behaviour. These interactions are insignificant and can be considered as irrelevant for the purpose of the diagram, however, they may still occur during the actual execution. The overall behaviour of the system does not change, whether the local transitions within *ignore* fragments occur or not. It allows a way of taking a perspective over an interaction.



(a) (b)

Figure 5.22: A sequence diagram with ignorance behaviour (a) and the corresponding CPN (b).

When transforming ignored behaviour of a SD, the corresponding CPN contains net transitions $t_{ignore-beg}$ and $t_{ignore-end}$ to synchronise the tokens at the beginning and the end of the fragment, as given by Rule 5.15. Since the

firing of the net transitions within the ignore behaviour does not make any change, we impose an equality between the places correspond to the minimum and maximum state locations within the fragment for a given instance in the SD (see Figure 5.22). Thus, the system states at the beginning and the end of the fragment are equivalent.

The following rule defines the transformation of an *ignore* fragment to a CPN representation.

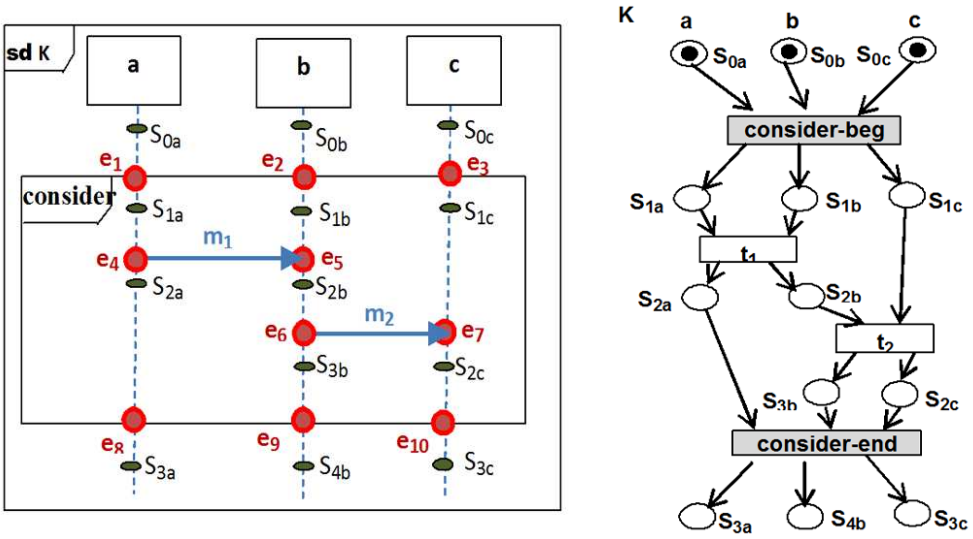
Rule 5.23 (ignore-Rule) *Let $x \in F$ be an interaction fragment in SD with $f(x) = (ignore, 1)$, and $i \in j(x, 1)$ be an arbitrary instance involved in the fragment. According to Rule 5.15 $s, s_1, s_1^{'}, s^{'} \in S_{int}^{i}$ are state locations for instance i, and the associated CPN contains places $s, s_1, s_1^{'}, s^{'} \in P$, transitions $t_{ignore-beg}, t_{ignore-end} \in T_n^{\neg l}$, and arcs $a_{i0}, a_{i1}, a_{i1}^{'}, a_{i0}^{'} \in A$. For an ignore fragment we have additionally the equality $s_1 = s_1^{'}$.*

Consider the SD with the ignorance behaviour and the corresponding CPN shown in Figure 5.22. SD_K contains a *ignore* interaction fragment with two local transitions $t_1, t_2 \in T$. For the instance $b \in j(x, 1)$, the state locations within the fragment are given by $\lambda_b(x, 1) = \{S_{1b}, S_{2b}, S_{3b}\}$ and $min_b(\lambda_b(x, 1)) = S_{1b}$, $max_b(\lambda_b(x, 1)) = S_{3b}$. By applying Rule 5.23 the corresponding CPN contains net transitions $ignore - beg, ignore - end \in T_n^{\neg l}$ and $t_1, t_2 \in T_n$ and places $S_{1b}, S_{2b}, S_{3b} \in P$ of colour b. To indicate the ignorance behaviour of the transitions t_1, t_2, we have the equality between the *min* and *max* places such that $S_{1b} = S_{3b}$.

5.4.10 Transformation of Consider Behaviour

The *consider* interaction fragment represents the possible behaviour that is intentionally included in the interaction. As defined by the UML standard [OMG, 2011a], the local transitions within a *consider* fragment are designated

to be relevant. The behaviour of this operator can be considered as similar to the default behaviour of an interaction and can often be omitted. However, even the transformation applies, the corresponding CPN for the *consider* interaction fragment behaviour can be obtained from Rule 5.15 and no further restrictions are necessary (Figure 5.23.



(a) (b)

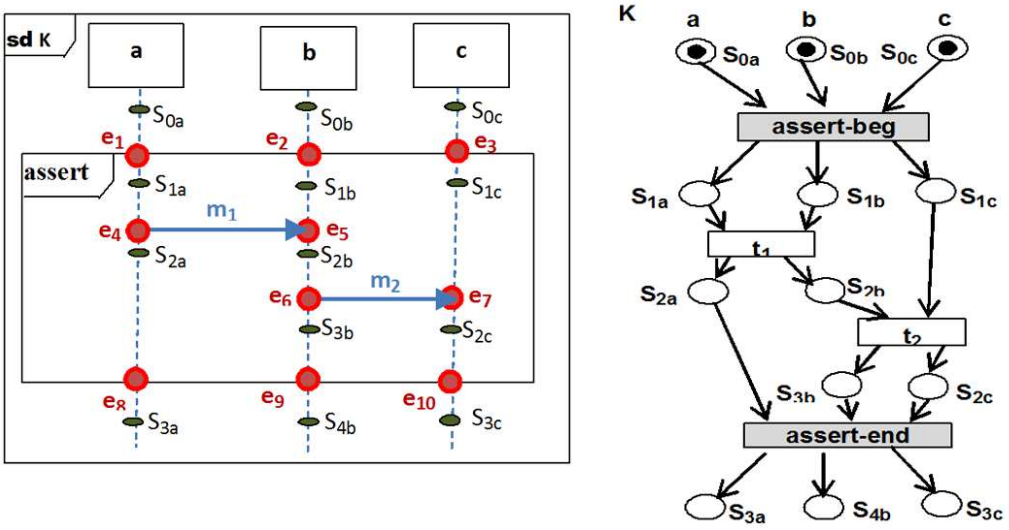
Figure 5.23: A sequence diagram with consider behaviour (a) and the corresponding CPN (b).

5.4.11 Transformation of Assertion Behaviour

The *assert* interaction fragment used in SDs specifies a mandatory interaction behaviour. The local transitions in an *assert* fragment indicates the only valid continuations. Since, the definition of *assert* is not well defined in the standard [OMG, 2011a], we assume that an assert fragment specifies the only valid interaction behaviour, and all the specified behaviour in the fragment *must* happen. That is, the interactions indicated by the local transitions within the fragment are required to execute, and only a part of the interactions is not

acceptable.

Applying Rule 5.15, an *assert* interaction fragment is transformed to a CPN by having $t_{assert-beg}$ and $t_{assert-end}$ net transitions to synchronise the token flow at the beginning and the end of the fragment. The behaviour of an *assert* fragment is represented in CPN by assigning an *incomplete* status to the places correspond to the state locations within the fragment, given by λ function, except the minimum and maximum state locations. A *complete* status is reached only when the control flow reaches the places before the net transition $t_{assert-end}$, after firing all the net transitions within the assertion behaviour. That is, we can ensure that all the transitions within the *assert* behaviour should happen in order to reach to a place with a *complete* status.



(a) (b)

Figure 5.24: A sequence diagram with assertion behaviour (a) and the corresponding CPN (b).

The following rule defines the transformation of an *assert* fragment to a CPN representation.

Rule 5.24 (assert-Rule) *Let $x \in F$ be an interaction fragment in SD with $f(x) = (assert, 1)$, and $i \in j(x, 1)$ be an arbitrary instance involved in the fragment. The corresponding CPN is obtained according to Rule 5.15. Additionally, for an assert fragment $\forall p_k \in P$ that correspond to the state locations given by $\lambda(x, 1) \setminus \{min(\lambda(x, 1) \cup max(\lambda(x, 1))\}$, $status(p_k) = \{incomplete\}$, $k \in \mathbb{N}$.*

For example consider the SD and the corresponding CPN shown in Figure 5.24. CPN_K contains places $S_{1b}, S_{2b}, S_{3b} \in P$ of the colour $b \in Sigma$ that correspond to the state locations $\lambda_b(x, 1) = \{S_{1b}, S_{2b}, S_{3b}\}$ for instance $b \in I$. By applying Rule 5.24, $status(S_{2b}) = \{incomplete\}$ and by default all other places are *complete*.

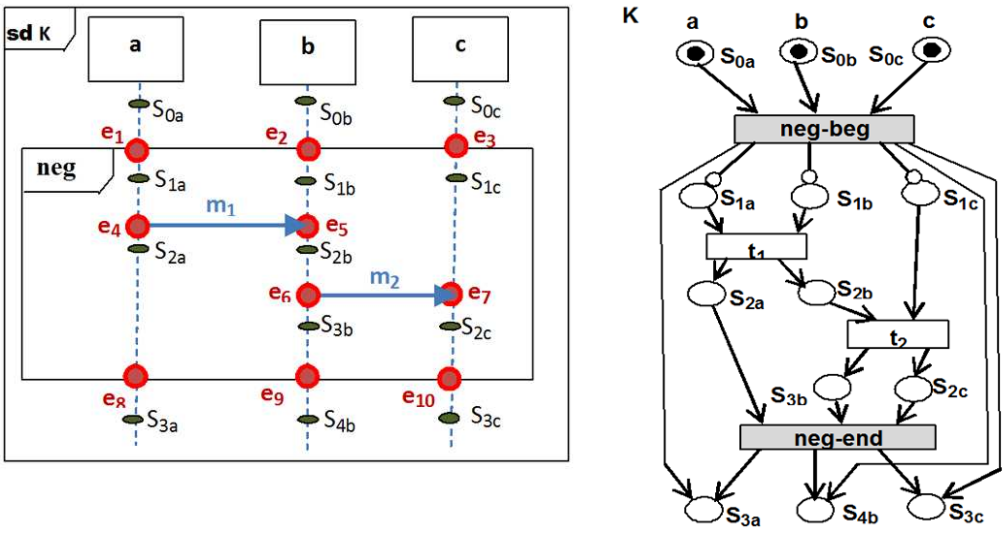
The definition indicates that the invalid traces are associated with only negate fragment, thus not associated with other fragments; and this contradicts the assert statement.

5.4.12 Transformation of Negative Behaviour

The *neg* operator is used in SDs, to specify forbidden interactions of a system. It represents an invalid trace and specifies a behaviour that must not occur. Thus, the expressive power of the *neg* interaction fragment supports the safety and security properties in a system specification.

In order to represent the *neg* behaviour in a CPN, we use inhibitor arcs to link the net transition $t_{neg-beg}$ corresponds to the beginning of the fragment, with the places correspond to the minimum state locations of the fragment (see Figure 5.25). Thus, tokens will not pass to the net transitions within the *neg* behaviour and this will prevent their firing. Additionally, we link the transition $t_{neg-beg}$ with the place corresponds to the state location outside the fragment, (given by the θ function). This allow continuing with the transitions

after the *neg* behaviour. Further, we assign an $unsafe$ status for all the places correspond to the state locations within the *neg* fragment, given by the λ function.



(a) (b)

Figure 5.25: A sequence diagram with negative behaviour (a) and the corresponding CPN (b).

The following rule defines the transformation of a *neg* fragment to a CPN representation.

Rule 5.25 (neg-Rule) *Let $x \in F$ be an interaction fragment in SD with $f(x) = (neg, 1)$, and $i \in j(x, 1)$ be an arbitrary instance involved in the fragment. According to Rule 5.15 $s, s_1, s_1^{'}, s^{'} \in S_{int}^{i}$ are state locations for instance i, and the associated CPN contains places $s, s_1, s_1^{'}, s^{'} \in P$, transitions $t_{neg-beg}, t_{nege-end} \in T_n^{\neg l}$, and arcs $a_{i0}, a_{i1}, a_{i1}^{'}, a_{i0}^{'} \in A$. For a neg fragment we have additionally an inhibitor arc $a_i^{''} \in A_{in}$ such that $node(a_i^{''}) = (t_{neg-beg}, s_1)$ and the arc a_{1i} is link such that $node(a_{i1}) = (t_{neg-beg}, s^{'})$. Also, $\forall s_k \in P$ that correspond to the state locations given by $\lambda(x, 1)$, $status(s_k) = \{unsafe\}$, $k \in \mathbb{N}$.*

Figure 5.25 shows a SD with negate behaviour and the corresponding CPN. Consider the instance $a \in I$ in the SD and the corresponding colour $a \in Sigma$ in the CPN. The CPN contain places $S_{1a}, S_{3a} \in P$ that correspond to the state locations such that $min(\lambda_a(x, 1)) = S_{1a}$ and $\theta_a(x) = S_{3a}$, respectively. The arc $a_1 \in A$ is linked such that $node(a_1) = (neg - beg, S_{3a})$ and there is an additional inhibitor arc $a_2 \in A_{in}$ such that $node(a_2) = (neg - beg, S_{1a})$. Thus the token do not pass to the net transitions t_1, t_2 and the negate behaviour is preserved.

5.5 Concluding Remarks

This chapter has described the model transformation framework used in this thesis, and a motivation as to why model transformation is useful from graphical modelling languages to formal models. The model transformation rules defined in this chapter are exogenous and based on the operational approach. The defined rules comply with the SD and CPN definitions given in Chapter 3 and Chapter 4, respectively. The defined rules have covered the transformations of the entire UML 2 SD elements including the behaviour of all the interaction fragments to the corresponding CPNs.

6 Complex Model Transformation

In this chapter, we consider ways of understanding the complexity of the interaction models and their transformations.

Software design models are rarely standalone and are generally connected to and depend upon other models or views of the system. The growth of large-scale and complex systems has resulted in software design models with a large number of interactions [Anda et al., 2009, France and Rumpe, 2007]. The development of such systems involves large collections of models for the same or different system perspectives [Kleppe et al., 2003,Vale and Hammoudi, 2009]. It is a challenge to compose various design models in a way that can facilitate them to function together as a system and are able to deliver required functionality. For example, a single model can be generated that gives a unified understanding about the entire system and enables end-to-end reasoning for properties that the system must satisfy. Model composition at the design stage is important to resolve issues that would not otherwise appear until the later stages of the development and when operationalise the system [Radjenovic and Paige, 2010].

Additionally, in a complex system there may be situations to check a property of the model against only a part of the behaviour. Here, we address this through the use of partial transformations. Partial transformation is of interest for local analysis and can be used to facilitate the understanding of a set of sub-interactions in a model [C.Baier and J.Katoen, 2008]. Conversely, partial transformations also support for the construction of specifications incrementally by combining previously developed models with new interactions that allows model reuse and analysis [Cuadrado et al., 2011].

Further, in model-to-model (M2M) transformation where the target model has variants, parametric transformation can be used to map the source model

to an intended target semantic model. We look into parametric transformations in the context of modelling and analysing systems with real-time and/or performance requirements.

Finally, we extend the defined transformations to model integration considering hierarchical aspects.

This chapter starts with addressing model composition between sequence diagrams (SDs) and coloured Petri nets (CPNs) considering the reference behaviour. Section 6.2 shows the applicability of partial and incremental transformations using rules defined in Section 6.1. Additionally, the parametric transformations that map SDs with timed and stochastic aspects to timed CPN (TCPN) and stochastic CPN (SCPN) models are described in Section 6.3. Further, Section 6.4 gives the model integration rules by considering hierarchical aspects of models. This includes defining transformation rules between SDs and (IODs) and between CPNs and hierarchical CPNs (HCPNs).

6.1 Model Composition

When modelling systems with a large number of interactions it is important to be able to decompose a large SD model into smaller units making use of an interaction-use (reference behaviour) or lifeline decomposition, so that each sub-model can be analysed separately. Conversely, it may be necessary to compose SD models to a single model to have a more global view of a system model. An important part of the M2M transformations in this thesis is to describe solutions for automatic model composition.

Sequence diagrams and CPNs allow composition and decomposition of models. SDs make this possible through *interaction-use* and *lifeline decomposition*. This section defines the transformation rules for the reference behaviour allowing partial synthesis of model transformation from SDs to CPNs. There

are several ways to transform complex SDs into CPNs as shown in Figure 6.1.

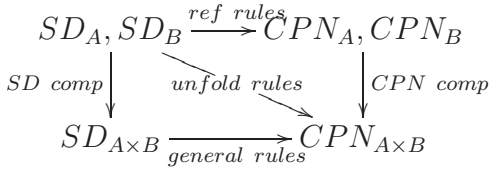


Figure 6.1: The transformation paths for SDs with decomposition mechanisms.

Given a sequence diagram SD_A with one or more references to SD_B, we can do one or more of the following:

- Transform both SDs into appropriate CPNs obtaining CPN_A and CPN_B where CPN_A is a complex CPN with some reference to CPN_B. These CPNs can be analysed directly or if intended, a composite CPN can be obtained through CPN composition rules (replace the occurrences of CPN_B in CPN_A);

- Compose SD_A and SD_B applying SD composition rules (replace the occurrences of SD_B in SD_A) and use basic rules to obtain a composite CPN;

- Apply the unfolding of SD_B in SD_A directly obtaining the composite CPN. A unique (up to bisimulation) CPN model $CPN_{A\times B}$ can be obtained from each path of transformations, i.e. the diagram of Figure 6.1 is preserved.

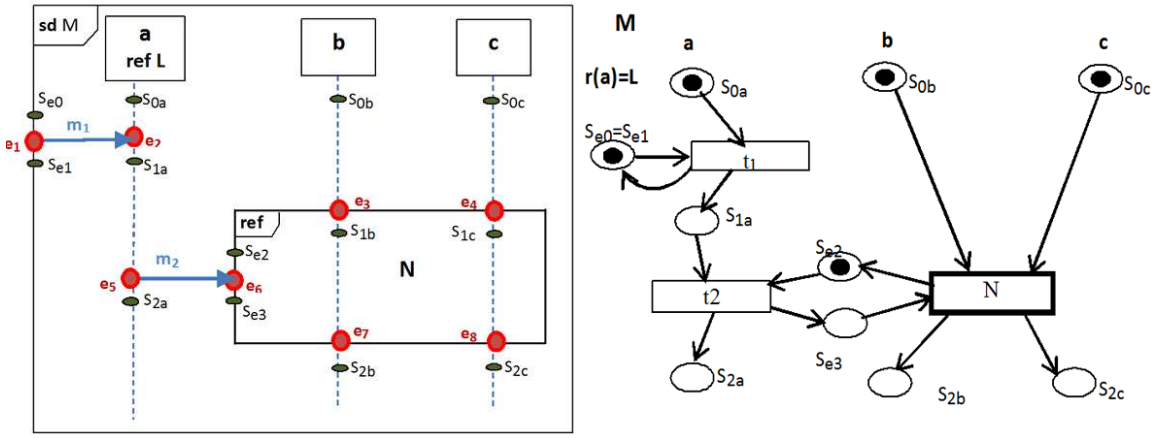
6.1.1 Model Composition with ref Rule

A SD with decomposition mechanism enables to represent a set of interactions in a separate diagram, allowing interactions to be reused in various ways. Recalling Chapter 3, Section 3.1.3, UML 2 decomposition mechanisms consist of interaction-use (ref fragment) and lifeline decomposition.

Recalling the interaction-use behaviour of a SD, there should be a separate SD with the same diagram name referred by the *ref* fragment. When composing the two diagrams, the *ref* fragment is replaced by the behaviour of the referred diagram.

In general transformation of a SD to a CPN (defined in Chapter 5), for each local transition in a SD there is a corresponding net transition in the CPN. Here, the net transition is labelled by the corresponding message label of the local transition.

The transformation of an interaction-use (*ref* fragment) in a SD to the corresponding CPN is as follows: The CPN includes an additional net transition to represent the behaviour abstract by the *ref* fragment, and labels that net transition by the referred diagram name given by the *ref* fragment.



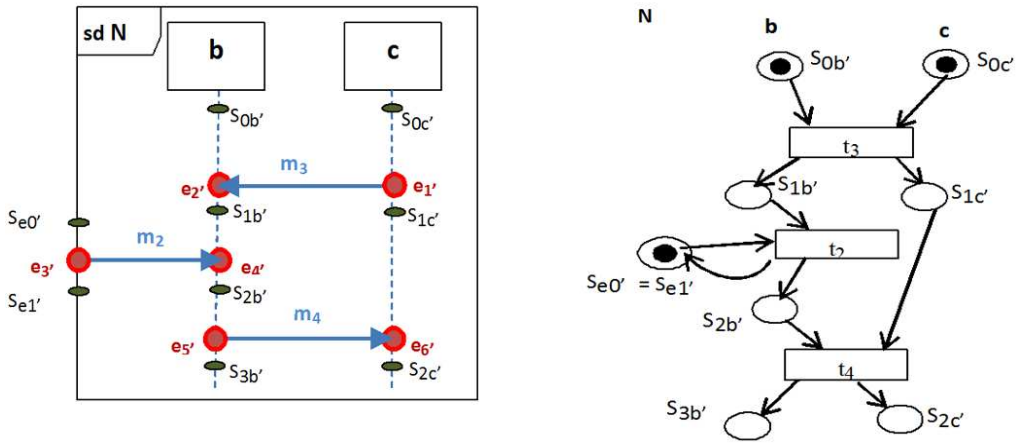
(a) (b)

Figure 6.2: The decomposition behaviour of a SD (a) and the corresponding CPN (b).

Here, the transition is in effect a substitution transition, which conveys the behaviour of the referenced CPN (Section 6.1.3, Section 6.1.4 and Section 6.4 describes more on this).

The following rule states the transformation of a *ref* interaction fragment in a SD to a CPN.

Rule 6.1 (Interaction-Use-Rule) *Let SD_A be a sequence diagram where $x \in F_A$ is such that $f(x) = (ref, 1)$ and $ref(x) = B$. For all $i \in j(x,1)$ instances involved in x let $\lambda_i(x,1) = \{s_{1i}\}$ and $\theta_i(x) = s_{2i}$. The corresponding CPN_A is such that $i \in \Sigma^+$, $s_{1i}, s_{2i} \in P_A$ with $c(s_{1i}) = c(s_{2i}) = i$ and there is an additional net transition $t \in T_{nA}$ such that $l(t) = B$ and there are arcs $a_{1i}, a_{2i} \in A$ such that $node(a_{1i}) = (s_{1i}, t)$ and $node(a_{2i}) = (t, s_{2i})$. If $i \in j(x,1) \cap I_A$ is an object instance, then for $e = min_i(g(x,1))$ with $e \in next_i(s_{0i})$, the matching places in P_A satisfy $s_{0i} = s_{1i} \in P_A$.*



(a) (b)

Figure 6.3: A referred sequence diagram by interaction-use (Figure 6.2) (a) and the corresponding CPN (b).

Consider SD_M and CPN_M shown in Figure 6.2 with an *interaction-use* behaviour. The interaction fragment ref in SD_M refers to SD_N shown in Figure 6.3. Applying Rule 6.1 to SD_M and by looking at the instances $b, v_2 \in j(x,1)$ involved in fragment x with $ref(x) = N$, we obtain the following as described in Chapter 3. For the object instance $b \in I_M$, the associated

state locations are determined by $\lambda_b(x,1) = \{S_{1b}\}$ and $\theta_b(x) = S_{2b}$. For the environment instance $v_2 \in I_M^+$, the associated state locations are determined by $\mu_{v_2}(m_3, e_6) = S_{3e}$, $\lambda_{v_2}(x,1) = \{S_{3e}\}$ and $\theta_{v_2}(x) = S_{2e}$.

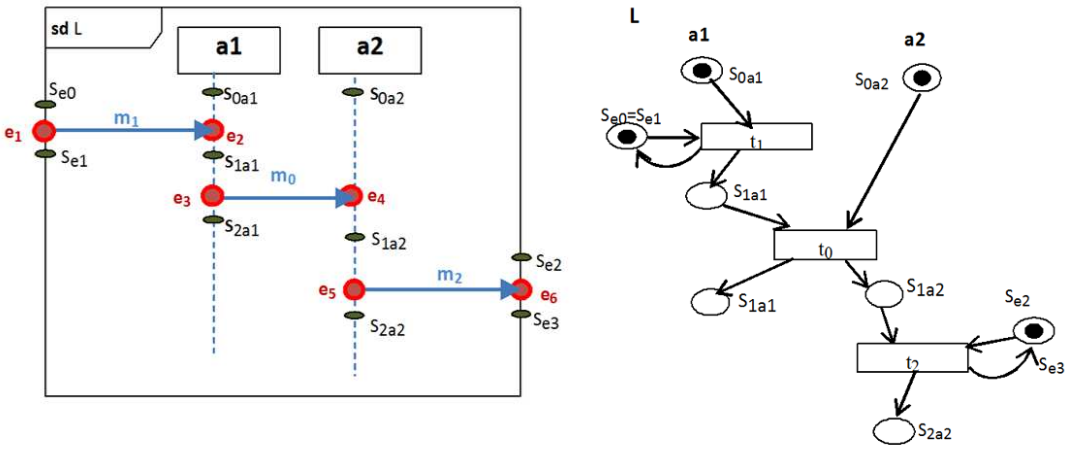
The CPN representation for an interaction-use is represented by an additional net transition $t_N \in T_n$ and is labelled with the diagram name N: $l(t_N) = N$. Thus, when executing the net transition t_N, it substitutes the referred diagram CPN_N (Substitution is defined in Section 6.1.3). The colours and places of the CPN correspond to the instances and state locations of the SD, as given by the main transformation rules in Chapter 5. According to Rule 6.1, for the colour b that corresponds to an object instance, we have the equality of places such that $S_{1b} = S_{0b}$. Further, there are arcs $a_{1b}, a_{2b} \in A$ such that $node(a_{1b}) = (S_{1b}, N)$ and $node(a_{2b}) = (N, S_{2b})$.

Consider the Figure 6.2, for the environment instance $v_2 \in I_M$, CPN_M contains places S_{e2}, $S_{e3} \in P$ correspond to the state locations $\lambda_{v_2}(x,1) = \{S_{e3}\}$ and $\theta_{v_2}(x) = S_{e2}$. These places are link with the net transition t_N such that $node(a_{1v_2}) = (S_{e3}, N)$ and $node(a_{2v_2}) = (N, S_{e2})$, where $a_{1v_2}, a_{2v_2} \in A$.

Since, $e_6 \in g_{v_2}(x)$ is a gate event which is associated with the local transition $t_2 = (e_5, m_2, e_6)$, we also have $\mu_{v_2}(m_1, e_6) = S_{e3}$ and $e_6 \in next_{v_2}(S_{e2})$. According to Rule 6.1 we have one arc connecting S_{e2} with the transition t_2 and another arc connecting the transition t_2 with S_{e3}. I.e. the arcs $a_{3v_2}, a_{4v_2} \in A$ link the corresponding net transitions $t_2 \in T_n$ with the places such that $node(a_{3v_2}) = (S_{e2}, t_2)$ and $node(a_{4v_2}) = (t_2, S_{e3})$.

Lifeline decomposition is another mechanism that enables diagram referencing. Here, an instance can refer the behaviour of a separate sequence diagram (e.g. instance a in SD_M of Figure 6.2 refers SD_L). The transformation of this behaviour to a CPN hardly affects the visual representation of the CPN model. The corresponding CPN can be obtained with normal rules adding only the

information that colour $a \in \Sigma$ is such that $r(a) = L$, where L is the referred diagram. When the two diagrams combings, the places and the transitions that are associated with the colour a are replaced by appropriate behaviour in the referred diagram L (this substitution is defined in Section 6.1.3).



(a) (b)

Figure 6.4: The referred sequence diagram by lifeline decomposition (Figure 6.2) (a) and the corresponding CPN (b).

The transformation of lifeline decomposition behaviour to a CPN is defined in the following rule.

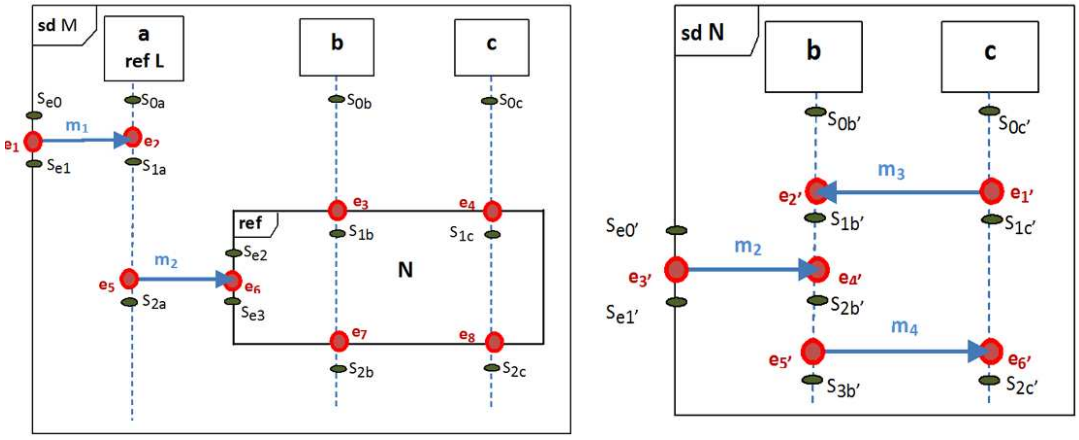
Rule 6.2 (Lifeline-Decomposition-Rule) *Let SD_A be a sequence diagram where $i \in I_A$ is such that $ref(i) = B$ and $B \in \mathcal{N} \setminus \{A\}$. In the corresponding CPN_A, $i \in \Sigma$ and $r(i) = B$.*

Figure 6.2 shows a SD with lifeline decomposition and the corresponding CPN. The lifeline decomposition of the instance $a \in I$ refers to SD_L shown in Figure 6.4 such that $ref(a) = L$. By applying Rule 6.2 the corresponding CPN contains a colour $a \in \Sigma$ with $r(a) = L$.

Recalling Figure 6.1, consider the transformation of SD_A that has a reference to SD_B. The reference behaviour is also reflects in the corresponding CPN_A that refers to CPN_B. So that the transformation with *ref rules* in this figure can be obtained by applying Rule 6.1 and Rule 6.2.

6.1.2 Model Composition with *unfold* Rule

This section describes direct transformation rules to compose a CPN from SDs with decomposition mechanisms. Here, we define *unfold rules* shown in Figure 6.1 considering the two cases: (1) composition with *interaction-use* and (2) composition with *lifeline decomposition*.



(a) (b)

Figure 6.5: A sequence diagram with reference behaviour (a) and the referred SD with interaction-use (b).

When a SD has a reference to another SD with an *interaction-use* (*ref* interaction fragment), the corresponding CPN that represents the combined behaviour of two SDs by replacing the interaction-use can be obtained as follows. Consider Figure 6.5 and Figure 6.6. For all instances (eg. $b \in I$) involved in the *ref* fragment in the SD and for the corresponding colours in the CPN,

we impose an equality between the CPN places that correspond to the state location before the beginning of the ref fragment and the initial state location of the referred SD ($S_{0b} = S_{0b'}$) , for a given instance. Similarly, there is equality between the places that correspond to the state location after the end of the ref fragment and the end state location of the referred SD ($S_{2b} = S_{3b'}$), for all the instances involved in the fragment. Following Rule 6.3 defines this transformation.

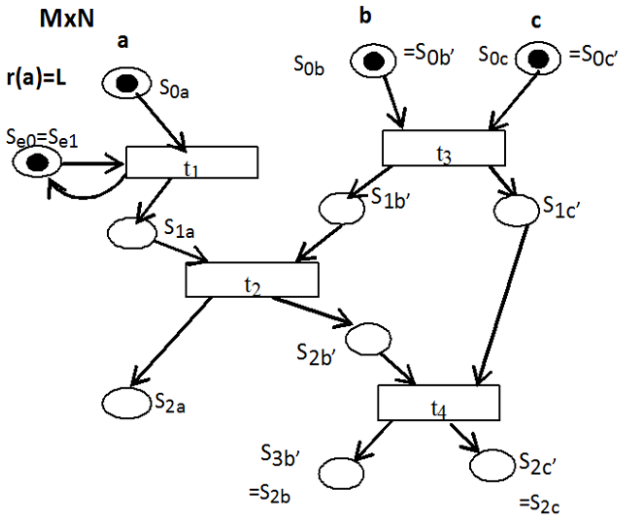


Figure 6.6: The corresponding CPN obtained from $SD_M \times SD_N$.

Rule 6.3 (Unfold-Rule:1 (with interaction-use)) *Let SD_A, SD_B be two sequence diagrams where $x \in F_A$ is such that $f(x) = (ref, 1)$ and $ref(x) = B$. For all $i \in j(x, 1)$ instances involved in x and by definition necessarily $i \in I_B^+$. Let $e_1, e_2 \in E_A$ denote the minimal and maximal event in $\overline{g(x)_i}$, in SD_A respectively, where $e_1 \in next_i(s)$ and $next_i(e_2) = \theta_i(x) = s'$. Also, let $s_1 \in S^i_{ini_B}$ and $s_2 \in S^i_{end_B}$ be the initial and end state locations of the instance i in the referred diagram SD_B.*

The corresponding $CPN_{(AxB)}$ contains $i \in \Sigma^+$, $s, s^{'}, s_1, s_2 \in P$. The unfolded CPN representation of SD_A and SD_B is obtained by imposing an equality between the corresponding places such that $s = s_1$ and $s^{'} = s_2$.

Additionally, if $\exists v \in j(x,1) \cap Env$, there are two corresponding local transitions in the two SDs that connect with a gate as the source or target event, respectively. For $e, e_v \in E_A$, let $t \in T_A$ such that $t = (e, m, e_v)$ or $t = (e_v, m, e)$ where $e_v = min_v(g(x,1))$ and the corresponding local transition $t^{'} \in T_B$: $t^{'} = (e_v^{'}, m, e^{'})$ or $t^{'} = (e^{'}, m, e_v^{'})$ for $e^{'}, e_v^{'} \in E_B$, where $e_v^{'} \in next_v(s_{0v})$ and $s_{0v} \in S_{ini_B}^v$ and $l(t) = l(t^{'}) = m$.

Then, there is an equality between the corresponding net transitions $t, t^{'} \in T_n$ such that $t = t^{'}$ and the unfolded CPN does not contain a colour for the corresponding environment instance: $v \notin \Sigma^+$ and does not contain the corresponding places for the colour v.

Consider SD_M with the interaction-use behaviour that refers to SD_N, shown in Figure 6.5. Figure 6.6 shows the corresponding CPN for the composition of the two SDs obtained by unfolding the SDs. Consider the object instance b in both diagrams. The state locations before the beginning of the ref fragment and after the end of the fragment in SD_M are given by S_{0b}, S_{2b} respectively. In SD_N, the initial and end state locations for object b are given by $S_{0b'}$ and $S_{3b'}$, respectively. By applying Rule 6.3, CPN_{MxN} is obtained with the colours $a, b, c \in \Sigma$ from both diagrams. For the colour b, the CPN has corresponding places $S_{0b}, S_{2b}, S_{0b'}, S_{3b'} \in P$, whereby $S_{0b} = S_{0b'}$ and $S2_b = S_{3b'}$.

Additionally, consider the local transition $t_2 = (e_5, m_2, e_6)$ in SD_M, that connects to the ref interaction fragment through a gate, and the corresponding local transition $t_2 = (e_3^{'}, m_2, e_4^{'})$ in SD_N. There is an equality between the two net transitions $t_2, t_2^{'} \in T_n$ in CPN_{MxN}, : $t_2 = t_2^{'}$. Further the CPN does not contain any place of the colour $v \in \Sigma^+$ that corresponds to the $v \in Env$.

When a SD refers to another SD using *lifeline decomposition*, the corresponding CPN for the composition of two SDs can be obtained as follows. The CPN contains colours and places for all the instances and state locations of the two SDs, except for the instance with the lifeline decomposition and the associated state locations. Further the net transitions correspond to the local transitions that are involved in the lifeline decomposition instance are replaced by the corresponding net transitions in the referred diagram. Places and net transitions are linked as expected.

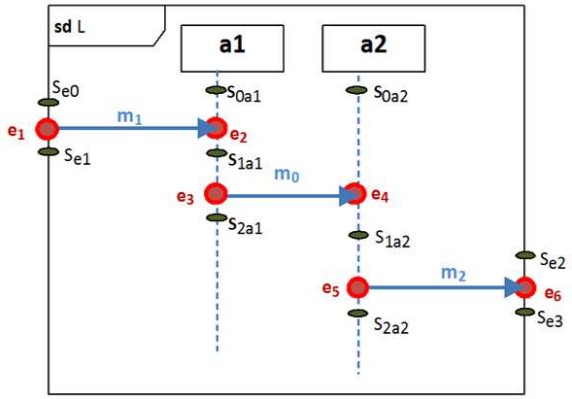


Figure 6.7: The referred SD from the lifeline decomposition of instance a in Figure 6.5.

Following Rule 6.4 defines the obtaining of a corresponding CPN from two SDs with a lifeline decomposition relation.

Rule 6.4 (Unfold-Rule:2 (with lifeline decomposition)) *Let SD_A, SD_B be two sequence diagrams where $i \in I_A$ is such that $ref(i) = B$ and $B \in \mathcal{N} \setminus \{A\}$. With the transformation τ, the unfolded CPN contains: colours $j^{'} \in \Sigma^{+}$, $j^{'} = \tau(j) : j \in (I_A^{+} \cup I_B^{+}) \setminus \{i_A\}$, places $s^{'} \in P : c(s^{'}) = j$, net transitions $t_k^{'} \in T_n$, $t_k^{'} = \tau(t_k) : t_k \in T_A \cup T_B$ for $k \in \mathbb{N}$.*

Let the source or target event of a local transition $t \in T_A$, is involved in the instance i with the lifeline decomposition such that $t = (e_1, m, e_2)$, $e_1 \in E_i$ and/or $e_2 \in E_i$. Then the referred SD also contains a local transitions with the same label: $\exists t' \in T_B$ where $l(t) = l(t')$. The corresponding CPN_{AxB} can be obtained by imposing an equality between the corresponding net transitions $t, t' \in T_n$: $t = t'$.

Here, if the source or target event is involved in an environment instance $v \in Env_A$, then the associated state locations be $s_1 \in S^v_{ini}$ and $s_2 \in S^v_{end}$. The corresponding state locations of the referred SD be $s'_1, s'_2 \in S^v$ for $v \in Env_B$. Then CPN_{AxB} is obtained by imposing an equality between the corresponding places such that $s_1 = s'_1$ and $s_2 = s'_2$.

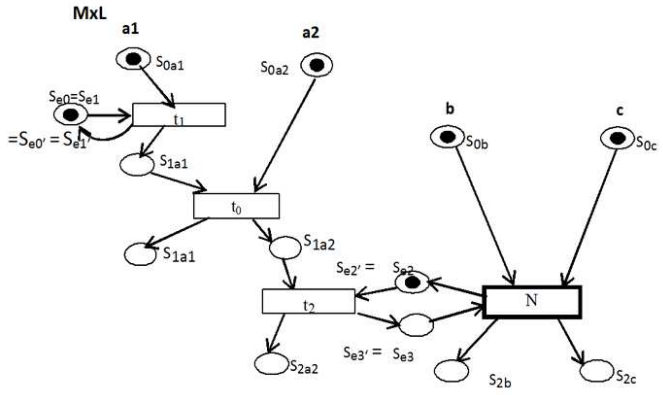


Figure 6.8: The corresponding CPN obtained from $SD_M \times SD_L$.

Consider SD_L in Figure 6.7 referred from SD_M in Figure 6.5 by the lifeline decomposition of instance a such that $ref(a) = L$. The local transitions $t_1 = (e_1, m_1, e_2)$, $t_2 = (e_5, m_2, e_6)$ in SD_M are involved with the lifeline decomposition instance and the corresponding local transitions in SD_L are $t'_1, t'_2 \in T_L$ with $l(t'_1) = m_1$ and $l(t'_2) = m_2$.

By applying Rule 6.4 the corresponding CPN shown in Figure 6.8 can be obtained by unfolding the two SDs. CPN_{MxL} contains colours $a1, a2, b, c, v_1, v2 \in \Sigma^+$ and does not contain $a \in \Sigma$. Also, there is an equality between the corresponding net transitions such that $t_1 = t_1^{'}$ and $t_2 = t_2^{'}$.

Further, consider the environment instance $v_1 \in (Env_M \cap Env_L)$ and the associated state locations $S_{e0}, S_{e1} \in S_M$ and $S_{e0}^{'}, S_{e1}^{'} \in S_L$. There is an equality between the corresponding places in CPN_{MxL} such that $S_{e0} = S_{e0}^{'}$ and $S_{e1} = S_{e1}^{'}$. (In default, for the places of the colour environment $S_{e0} = S_{e1}$ by Rule 5.7 for a gate event).

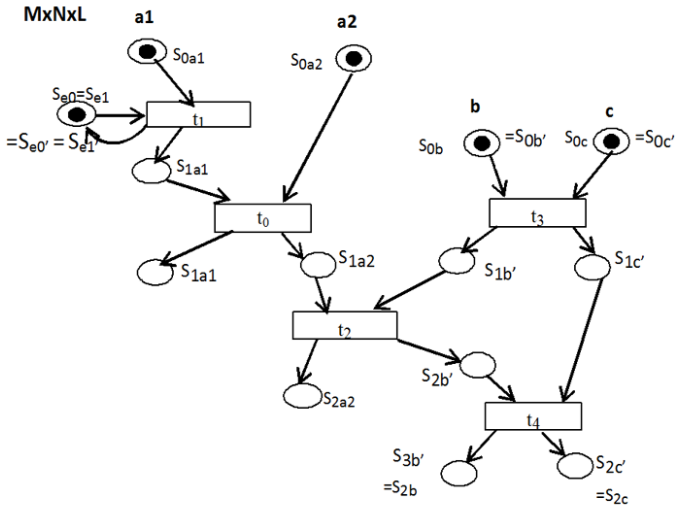


Figure 6.9: The corresponding CPN obtained from $SD_M \times SD_N \times SD_L$.

Consider the CPN model shown in Figure 6.9. CPN_{MxNxL} represents the composition of sequence diagrams SD_M, SD_N (in Figure 6.5), and SD_L (in Figure 6.7), obtained by applying the unfolded rules, Rule 6.3 and Rule 6.4. In CPN_{MxNxL}, the interaction-use fragment is replaced by the places and transitions correspond to SD_N. There is an equality between the net transitions correspond to the corresponding local transitions, and ignores the places

correspond to the environment state locations in between.

When transforming the lifeline decomposition behaviour, the CPN does not contain a colour $a \in \Sigma$ that correspond to the instance involved in the lifeline decomposition. During the transformation process, the instance with the lifeline decomposition and the associated state locations are replaced by the corresponding instances and state locations of the referred diagram, thus the places and the net transitions.

6.1.3 Model Composition with *CPNcomp* Rule

CPNs with reference behaviours can be composed to a separate CPN model that reflects the same behaviour as the source CPN models. The decomposition mechanisms associate with CPN models correspond to *interaction-use*and *lifeline decomposition* behaviours represent in SDs.

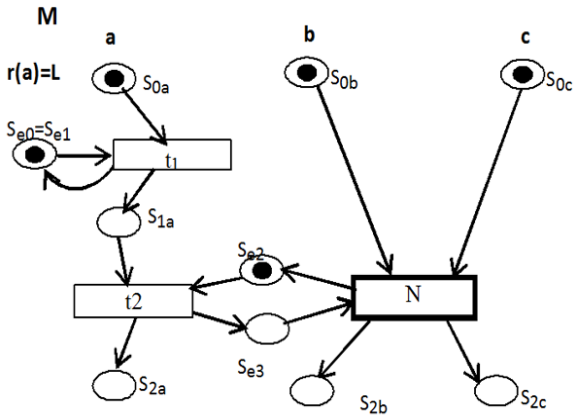
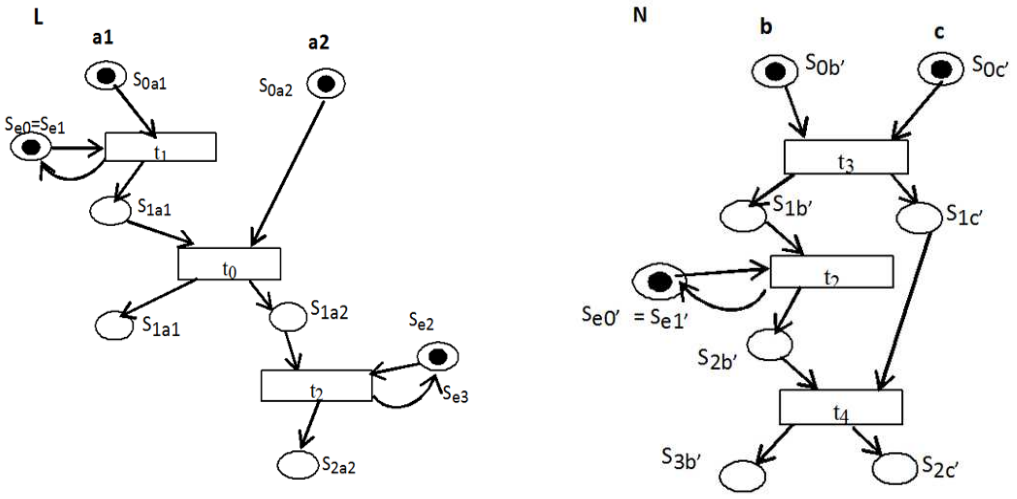


Figure 6.10: A CPN with reference behaviour.

Here, we address *CPN comp* transformation rules (as described in Figure 6.1) that compose a model CPN_{AxB} from the models CPN_A and CPN_B, where CPN_A has a reference to CPN_B, either by colours or net transitions.



(a) (b)

Figure 6.11: The referred CPNs from CPN_M(Figure 6.10) by colour reference (a) and transition reference (b)

When a colour $i \in \Sigma$ in a model CPN_A has a reference to another model CPN_B: $r(i) = B$, the composition of the two models can be obtained as follows. CPN_{AxB} model contains colours of the both models, except for the colour that has a reference to another model : $j \in (\Sigma_A^+ \cup \Sigma_B^+) \setminus \{i_A\}$. Thus the composite model does not contain any place of the colour i. Further, the net transitions linked with the places of the colour i are replaced by the corresponding net transitions and the linked places in the referred model. Additionally, if the two models contain environment colours, then there is an equality between the corresponding places of that colour.

Figure 6.10 shows a CPN model with two reference behaviours, (1) CPN_M contains a colour $a \in \Sigma$ that refers to another model such that $r(a) = L$, and the corresponding CPN_L is shown in Figure 6.11(a), (2) CPN_M has a net transition $t_r \in T_n$ with a label to another CPN such that $l(t_r) = N$ and the referred model CPN_N is shown in Figure 6.11(b).

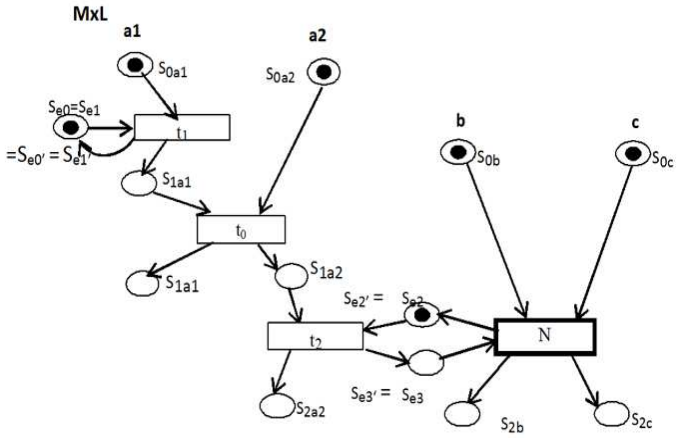


Figure 6.12: The composition of CPN_M and CPN_L.

From CPN_M with $r(a) = L$ and CPN_L, the composition of the two models can be obtained, as shown in CPN_{MxL} in Figure 6.12. CPN_{MxL} has object colours from the union of the colours from the source models, except for the colour that has a reference to $CPN_L : a1, a2, b, c \in \Sigma$ and the places of those colours and all the net transitions. CPN_{MxL} contains only one net transition for each corresponding transition by imposing an equality.

Consider the net transitions $t_{1_{(M)}}, t_{2_{(M)}} \in T_{n_M}$ and the corresponding $t_{1_{(L)}}, t_{2_{(L)}} \in T_{n_L}$ and there is an equality : $t_{1_{(M)}} = t_{1_{(L)}}$ and $t_{2_{(M)}} = t_{2_{(L)}}$. Additionally, when the source model (here, CPN_M) contains places of the environment colour, the composite model impose an equality between those places. Consider the places $S_{e0_{(M)}} \in P_M$ and $S_{e0_{(L)}} \in P_L$. There is an equality between these places in $CPN_{MxL} : S_{e0_{(M)}} = S_{e0_{(L)}}$. Other places and transitions are linked as given by the source models.

Rule 6.5 defines this transformation of two CPNs referred by a colour to a single CPN with the same behaviour.

Rule 6.5 (CPN-Composition-Rule:1) *Let CPN_A, CPN_B be two models with a colour reference: $r(i) = B$, for $i \in \Sigma$. Then the compositional model CPN_{AxB} contain colours $j \in (\Sigma_A^+ \cup \Sigma_B^+) \setminus \{i_A\}$, places $p \in P_A \cup P_B$ where $c(p) = j$, net transitions $t \in (T_{n_A} \cup T_{n_B})$ and the arcs that link places and net transitions with the node function as expected.*

Additionally, let $t_1 \in T_{n_A}$ and $t_2 \in T_{n_B}$ be two corresponding net transitions in the two models: $l(t_1) = l(t_2) \in T_{n_A} \cup T_{n_B}$. The composite model CPN_{AxB} is formed by imposing an equality between the net transitions: $t_1 = t_2$.

Similarly, let $p_1 \in P_A$ and $p_2 \in P_B$ be two corresponding places of the colour environment: $c(p_1) = c(p_2) = v \in \mathcal{E}$, CPN_{AxB} has equality $p_1 = p_2$.

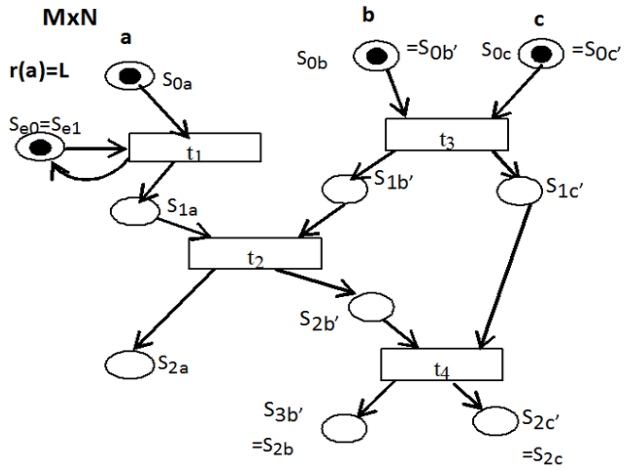


Figure 6.13: The composition of CPN_M and CPN_N.

When a CPN model refers to another CPN by the label of a net transition (let say, composite net transition), the corresponding compositional model is obtained as follows: The composite model contains the union of the colours, places, net transitions correspond to both source CPNs, except for the net transition with the label to the referred model and the places of the colour environment that are linked with that transition.

Further, we impose an equality between the source and the target places of that composite net transition, with the initial and end places of the referred model, respectively, for each colour. Additionally, there is an equality between the net transitions in the two CPNs, that has the same label and connected with a place of the colour environment.

Consider CPN_{MxN} shown in Figure 6.13. This composite model is obtained by the source models CPN_M (Figure 6.10) and CPN_N (Figure 6.11). CPN_M has a net transition $t_r \in T_{n_M}$: $l(t_r) = N$. Take a colour $b \in \Sigma_M \cap \Sigma_N$. The places $S_{0b}, S_{2b} \in P_M$ are the source and target places of the net transition t_r. In CPN_N the places $S_{0b'}, S_{3b'} \in P_N$ are the initial and end places of the colour b. Since the net transition t_r is replaced by CPN_N when obtaining the composite model, in CPN_{MxN}, $t_r \notin T_{n_{(MxN)}}$ and there is an equality between the places such that $S_{0b} = S_{0b'}$ and $S_{2b} = S_{3b'}$.

Additionally, the net transition t_r is linked with places $S_{e2}, S_{e3} \in P_M$ where $c(S_{e2}) = c(S_{e3}) = v \in \mathcal{E}_M$. Also, the net transition $t_{2_{(M)}} \in T_{n_M}$ has a connection with t_r via the places of the environment. The referred CPN_N also contains corresponding net transition $t_{2_{(N)}} \in T_{n_N}$ and the places $S_{e0'}, S_{e1'} \in P_M$ where $c(S_{e2'}) = c(S_{e3'}) = v \in \mathcal{E}_N$.

The composite model CPN_{MxN} does not contain places for the corresponding environment colour : $S_{e2}, S_{e3}, S_{e0'}, S_{e1'} \notin P_{MxN}$, and there is an equality between the net transition with the same label : $t_{2_{(M)}} = t_{2_{(N)}}$. All other places and net transitions are linked in the same manner as indicated by the source models.

Rule 6.6 defines this transformation of two CPNs referred by a net transition to a single CPN with the same behaviour.

Rule 6.6 (CPN-Composition-Rule:2) *Let CPN_A, CPN_B be two CPNs where $t_r \in T_{n_A}$ such that $l(t_r) = B$. In CPN_A, for each colours $i \in \Sigma$*

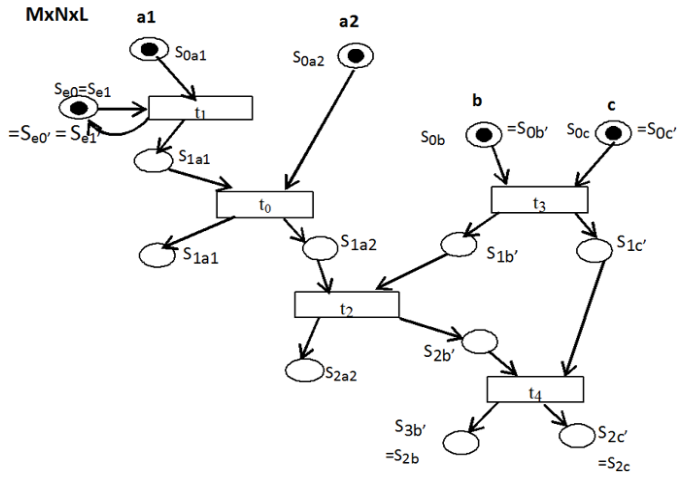


Figure 6.14: The composition of CPN_M, CPN_N and CPN_L.

involved with the net transition t_r *let* $p_1, p_2 \in P_A$ *and arcs* $a_{i1}, a_{i2} \in A$ *such that* $node(a_{i1}) = (p_1, t_r)$ *and* $node(a_{i2}) = (t_r, p_2)$. *For the corresponding colour in* CPN_B *let* $p_1^{'}, p_2^{'} \in P_B$. *The composition of* $CPN_{(AxB)}$ *can be obtained by imposing an equality between the places such that* $p_1 = p_1^{'}$ *and* $p_2 = p_2^{'}$ *has does not contain the net transition:* $t_r \notin T_n$.

Case I: If t_r *is linked with a place of the colour environment, then let* $t_1 \in T_{n_A}$ *and* $p_{e1}, p_{e2} \in P_A$ *where* $c(p_{e1}) = c(p_{e2}) = v \in \mathcal{E}_A$, *and let* $t_1^{'} \in T_{n_B}$ *and* $p_{e1}^{'} = p_{e2}^{'} \in P_B$ *where* $c(p_{e1}^{'}) = c(p_{e2}^{'}) = v \in \mathcal{E}_B$ *and* $l(t_1) = l(t_1^{'})$. *Let these places and net transitions are link by arcs* $a_{e1}, a_{e2}, a_{e3}, a_{e4} \in A_A$ *and* $a_{e5}, a_{e6} \in A_B$ *such that* $node(a_{e1}) = (p_{e1}, t_1)$, $node(a_{e2}) = (t_1, p_{e2})$, $node(a_{e3}) = (p_{e2}, t_r)$, $node(a_{e4}) = (t_r, p_{e1})$, $node(a_{e5}) = (p_{e1}^{'}, t_1^{'})$, $node(a_{e6}) = (t_1^{'}, p_{e1}^{'})$.

The compositional model CPN_{AxB} *contains colours* $i \in \Sigma_A^+ \cup \Sigma_B^+ \setminus \{v_A, v_B\}$, *places* $p \in P_A \cup P_B \setminus \{p_e\}$ *where* $c(p_e) = v$, *net transition* $t \in T_{n_A} \cup T_{n_B} \setminus \{t_r\}$ *and the arcs given by the node function. Additionally, there is an equality between the net transitions with the same label:* $t_1 = t_1^{'}$.

By applying both Rule 6.5 and Rule 6.6 we can obtain the composition of all three models CPN_M, CPN_N and CPN_L as given by CPN_{MxNxL} in Figure 6.14. This composite model can be obtained by removing the colours, net transitions that have a reference to another model and the places of the environment colour, that link with the removed net transition, and by imposing an equality between the corresponding net transitions that have the same label.

6.1.4 Model Composition with *SDcomp* Rule

Sequence diagram composition rules can be defined on a SD with decomposition mechanisms to obtain a detail SD that shows all the referred interactions. This section describes rules for the path *SD comp* shown in Figure 6.1, considering the two decomposition mechanisms, *lifeline decomposition* and *interaction-use*.

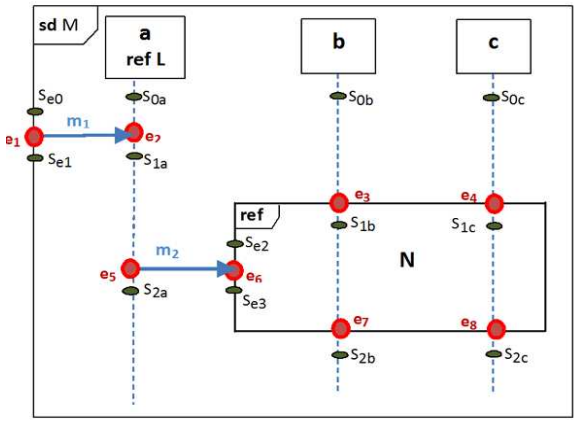
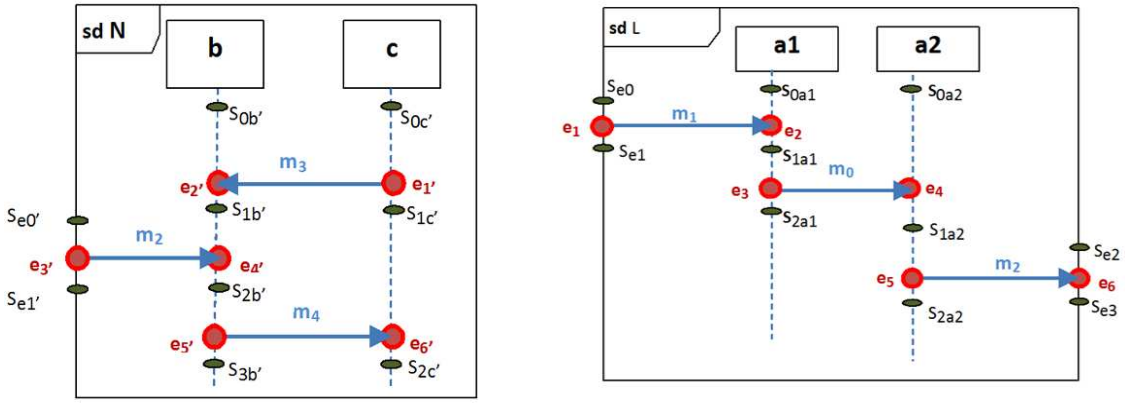


Figure 6.15: A sequence diagram with reference behaviour.

When a SD contains an instance with *lifeline decomposition* that refers to another SD, the behaviour of the two models can be composed to a single SD by detailing the reference behaviour. The transformation of the abstract SD and the referred SD to a composite SD can be obtained as follows. The

composite SD contains the union of the elements of the both models, except for the instance with the lifeline decomposition and all the events and state locations along that lifeline. Instead the new SD replaces with the elements of the referred SD.

Further, all the local transitions connected with the events along the removed instance as the source or target event, are substituted by the corresponding local transitions with the same message label, in the referred SD. Additionally, if there are state locations and events that belong to environment instances are linked with those common local transitions, then there is an equality between these state locations and events with the corresponding state locations and events in the referred SD, respectively.



(a) (b)

Figure 6.16: Referred sequence diagrams with interaction-use (a) lifeline decomposition (b).

Consider SD_{ML} shown in Figure 6.17, which is the composition of the diagram SD_M with lifeline decomposition (Figure 6.15) and the referred diagram SD_L (Figure 6.16(b)). The instance $a \in I_M$ is referred to SD_L using lifeline decomposition such that $ref(a) = L$. Here, for all the local transitions

$t_1, t_2 \in T_A$ associated with the instance a, there are corresponding local transitions $t_1, t_2 \in T_B$ in the referred diagram with the same message label. The combined diagram SD_{ML} contains the union of the elements in the source diagrams, expect for the instance a and all the associated events and state locations. Further, there is an equality between the local transitions that are common to the source diagrams.

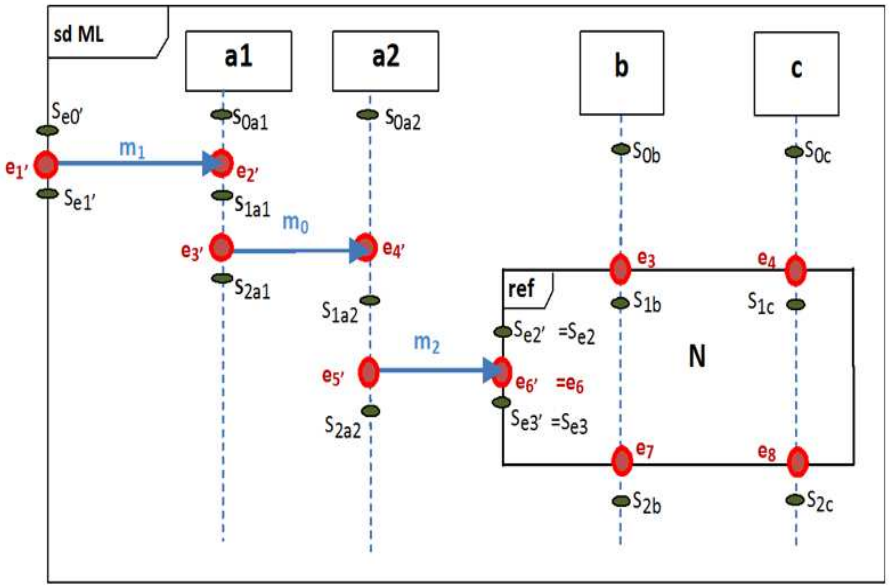


Figure 6.17: A sequence diagram obtained from SD_M and SD_L.

Additionally, SD_M contains events $e_1 \in E_{v_1}$, $e_6 \in E_{v_2}$ and state locations $S_{e0}, S_{e1} \in S^{v_1}$, $S_{e2}, S_{e3} \in S^{v_2}$. SD_L contains corresponding events $e_1^{'} \in E_{v_1}$, $e_6^{'} \in E_{v_2}$ and the associated state locations $S_{e0'}, S_{e1'} \in S^{v_1}$, $S_{e2'}, S_{e3'} \in S^{v_2}$ that belong to the environment instances $v_1, v_2 \in Env$. In the composite model, SD_{ML}, there is an equality between the corresponding events and state locations of the environment type: $e_1 = e_1^{'}$, $e_6 = e_6^{'}$, $S_{e0} = S_{e0'}$, $S_{e1} = S_{e1'}$, $S_{e2} = S_{e2'}$, $S_{e3} = S_{e3'}$.

Following Rule 6.7 defines the transformation of two SDs referred by a

lifeline decomposition to a single SD with the same behaviour.

Rule 6.7 (SD-Composition-Rule:1(lifeline decomposition)) *Let SD_A, SD_B be two sequence diagrams where $i \in I_A$ is such that $ref(i) = B$ and $B \in \mathcal{N} \backslash \{A\}$. The composite SD_{AxB} can be obtained by $(SD_A \backslash \{i, s_i, e_i\}) \cup SD_B$ where $s_i \in S_A^i$, $e_i \in E_{i_A}$.*

For all the local transitions $t \in T_A$ that are involved with the events involved with the lifeline decomposition instance i, as the source or target event, the referred SD contains corresponding local transitions with the same label: $\exists t^{'} \in T_B$ where $l(t) = l(t^{'}) = m$. SD_{AxB} can be obtained by imposing an equality between the local transitions $t = t^{'}$.

Additionally, if SD_A contains an environment instance $v \in Env_A$ and if there is an event $e_v \in E_A^v$ as a source or target event of t: $t = (e_v, m, e)$ or $t = (e, m, e_v)$ then the associated state locations $s_{v0}, s_{v1} \in S_A^v$ are such that $\mu(m, e_v) = s_{v1}$ and $e_v \in next_v(s_{v0})$. The referred SD_B has corresponding events and state locations $e_v^{'} \in E_B^v$, $s_{v0}^{'}, s_{v1}^{'} \in S_B^v$ for $v \in Env_B$. Then the composite SD_{AxB} is obtained by imposing an equality between the corresponding events and state locations such that $e_v = e_v^{'}$, $s_{v0} = s_{v0}^{'}$ and $s_{v1} = s_{v1}^{'}$.

When a SD refers a set of interactions within another SD using a ref interaction fragment, a single composite SD with the entire behaviour of interactions can be obtained as follows. The composite SD contains the union of the elements from the both, i.e., the SD with the abstract representation and the SD with the referred behaviour, except for the ref interaction fragment and the events and state locations involved in the interaction fragment. The composite SD is obtained by imposing an equality between the state locations correspond to the state location before the beginning of the ref fragment and the initial state location of the referred SD, for a given instance involved in the fragment. Similarly, there is equality between the state locations, which

correspond to the state location after the end of the ref fragment and the end state location of the referred SD, for all the instances involved in the fragment.

If a local transition connects to the ref interaction fragment through a gate event, then the referred SD also contains a corresponding local transition with the same message label. In this case, the composite SD is obtained by imposing an equality between these local transitions and removing the associated events and state locations that belongs to the environment instance.

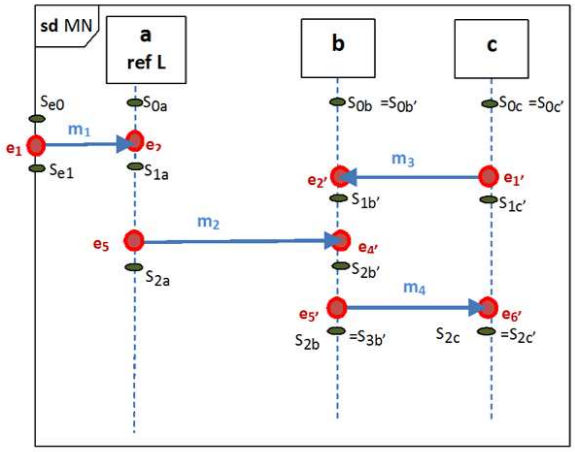


Figure 6.18: A sequence diagram obtained from SD_M and SD_N.

Consider, Figure 6.18 that represents the composition of the diagram SD_M with the interaction-use behaviour and the referred diagram SD_N that are shown in Figure 6.15 and Figure 6.16(a), respectively. The ref interaction fragment $x \in F_M$ in SD_M refers to the diagram SD_N such that $ref(x) = N$, $f(x) = (ref, 1)$ and $b, c, v \in j(x)$, $b, c, v \in I_N^+$.

Consider the instance b. In SD_M, the state locations before the beginning of the ref fragment and after the end of the fragment are given by S_{0b}, S_{2b}, respectively. In SD_N, the initial and end state locations for instance b are given by $S_{0b'}$ and $S_{3b'}$, respectively.

The composite model SD_{MN} can be obtained with the union of the elements

of the source models, except for the interactions fragment x and the associated minimum and maximum events and state locations of the fragment. Further, there is an equality between the state locations $S_{0b}, S_{2b}, S_{0b'}, S_{3b'} \in S_{MN}$ of instance b of SD_{MN} such that $S_{0b} = S_{0b'}$ and $S_{2b} = S_{3b'}$.

Additionally, consider the local transition $t_2 = (e_5, m_2, e_6) \in T_M$ connects to the fragment x via a gate and the associated $e_6 \in E^v$ and $S_{e2}, S_{e3} \in S_v$ where $v \in Env_M$. The corresponding local transition $t_2^{'} = (e_{3'}, m_2, e_{4'}) \in T_N$ where $e_{3'} \in E^v$ and $S_{e0'}, S_{e1'} \in S_v$ for $v \in Env_N$. In the composite model SD_{MN} there is an equality between the local transitions connect via the gate such that $t_2 = t_2^{'} = (e_5, m_2, e_{4'})$ and $v \notin I^+_{MN}$.

Following Rule 6.8 defines the transformation of two SDs referred by an interaction-use to a single SD with the same behaviour.

Rule 6.8 (SD-Composition-Rule:2 (interaction-use)) *Let SD_A, SD_B be two sequence diagrams where $x \in F_A$ is such that $f(x) = (ref, 1)$ and $ref(x) = B$. For all $i \in j(x,1)$ instances involved in x and by definition necessarily $i \in I^+_B$. Let $e_1, e_2 \in E_A$ denote the minimal and maximal event in $\overline{g(x)_i}$, in SD_A respectively, where $e_1 \in next_i(s)$ and $next_i(e_2) = \theta_i(x) = s^{'}$. Also, let $s_1 \in S^i_{ini_B}$ and $s_2 \in S^i_{end_B}$ be the initial and end state locations of the instance i in the referred diagram SD_B. The composite representation $SD_{(AxB)} = (SD_A \setminus \{x, e_1, e_2\}) \cup SD_B$ and there is an equality between the corresponding state locations such that $s = s_1$ and $s^{'} = s_2$.*

Additionally, if $\exists v \in j(x,1) \cap Env$, for $e_v, e \in E_A$ and $e_v^{'}, e^{'} \in E^v_B$ there are two corresponding local transitions that connect with a gate as the source or the target event, in the two SDs, respectively. Let $t \in T_A$ with the relevant gate event such that $t = (e, m, e_v)$ or $t = (e_v, m, e)$ where $e_v = min_v(g(x,1))$ and the corresponding $t^{'} \in T_B$: $t = (e_v^{'}, m, e^{'})$ or $t = (e^{'}, m, e_v^{'})$ where $e_v^{'} \in next_v(s_{0v})$, $s_{0v} \in S^v_{ini_B}$ and $l(t) = l(t^{'}) = m$. Then, there is an equality between the

corresponding local transitions in $SD_{AxB} : t = t'$ *and the composite SD does not contain state locations and events that belongs to the environment instance:* $v \notin \Sigma^+$.

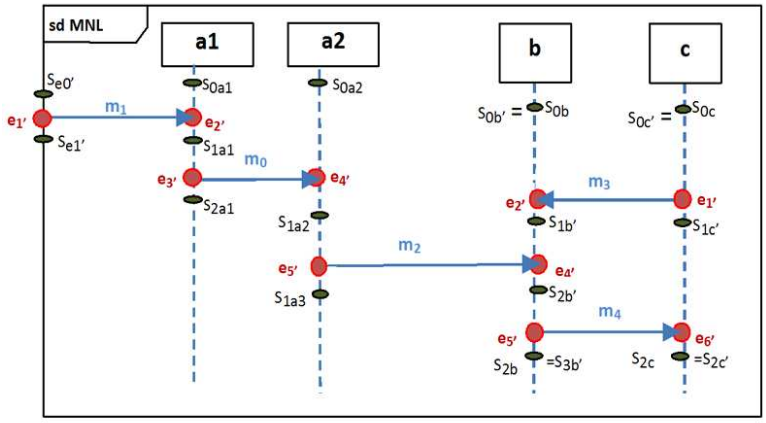


Figure 6.19: A sequence diagram combining SD_M, SD_N and SD_L.

The composite diagram SD_{MNL} shown in Figure 6.19 can be obtained as follows: (1) Applying Rule 6.7 and Rule 6.8 for the diagrams SD_M (Figure 6.15) that refers to both SD_L and SD_N (Figure 6.16) using *lifeline decomposition*and *interaction-use* behaviour, respectively. (2) Applying Rule 6.8 for the diagram SD_{ML} in Figure 6.17 that refers SD_N using *interaction-use*. (3) Applying Rule 6.7 for the diagram SD_{MN} in Figure 6.18 that refers to SD_L using *lifeline decomposition*.

Further, by considering the traces of the models, we can obtain behaviourally equivalent models using model composition. For example following diagrams are behaviourally equivalent: (1) $SD_M \otimes SD_N = SD_{MN}$, (2) $SD_M \otimes SD_L = SD_{ML}$, (3) $SD_M \otimes SD_N \otimes SD_L = SD_{MNL}$, (4) $SD_{MN} \otimes SD_L = SD_{MNL}$, (5) $SD_{ML} \otimes SD_N = SD_{MNL}$, where a common trace $m_1 \cdot m_0 \cdot m_3 \cdot m_2 \cdot m_4$ can be obtained from the composition of diagrams that are equivalent to SD_{MNL}.

6.1.5 Model Composition with *general* Rules

This section recalls the deriving of a CPN from a SD using *general rules* defined in Section 5.2.

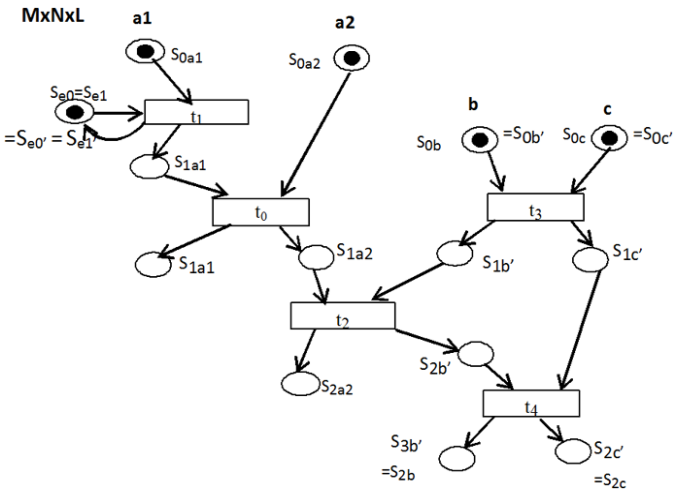


Figure 6.20: The corresponding CPN for SD_{MNL}.

For example, consider SD_{MNL} in Figure 6.19 and the corresponding CPN shown in Figure 6.20. SD_{MNL} contains $a1, a2, b, c, v \in I^+$, $t_1, t_0, t_3, t_2, t_4 \in T$ and the associated state locations. CPN_{MNL} with corresponding colours, places, net transitions and arcs can be obtained by applying the general rules defined in Section 5.2.

Recall the paths described in Figure 6.1. Here, the composite CPN models are obtained using the path *unfold rules*(Rule 6.3 and Rule 6.4 in Section 6.1.2) are same as the models obtained following the union of paths *ref rules*(Rule 6.1 and Rule 6.2 in Section 6.1.1) and *CPN comp*(Rule 6.5 and Rule 6.6 in Section 6.1.3). Also the same set of models can be obtained by the union of paths *SD comp* (Rule 6.7 and Rule 6.8 in Section 6.1.4) and *general rules* (SD to CPN transformation rules in Section 5.2).

For example, an identical CPN_{MNL} model can be obtained by the composition of $SD_M \otimes SD_N \otimes SD_L$ following the paths (1) *ref rules* $\cup$ *CPN comp* (2) *SD comp* $\cup$ *general rules* (3) *unfold rules*. I.e. a behaviourally equivalent model can be obtained following either of the path combinations.

Further, Rules described in this section supports for the transformations with hierarchical aspects of the models. For example, in Figure 6.1, *SD comp* rules enable to compose SD_{MN}, SD_{ML} and SD_{MNL} using SD_M with the decomposition mechanisms that refers SD_N and SD_L. Similarly, by applying *CPN comp* rules to CPN_M with the reference behaviour, together with the referred models, CPN_N and CPN_L, the composite models CPN_{MN}, CPN_{ML} and CPN_{MNL} can be obtained. Further, the *unfold-rules* are used to obtain the composite CPN models directly from SD_M that refers SD_N and SD_L.

6.2 Partial and Incremental Transformation

When a software system is modelled with a large number of interactions, there may be situations to analyse a property of the model concerning only a part of the behaviour. This section presents a formal approach for partial transformation of scenario-based specifications, which is a powerful constructs in sequence diagrams that enables incremental modular transformation.

Here, partial scenarios are captured as individual SDs and transformed to the corresponding CPNs. Partial transformation is of interest for local analysis, hence to get a better understanding of the sub model. Also, this supports an incremental development approach where interaction specifications are built incrementally and combined with previous iteration models. Figure 6.21 shows an overview of partial and incremental transformations. This section extends the described model transformation approach, for partial model transformation.

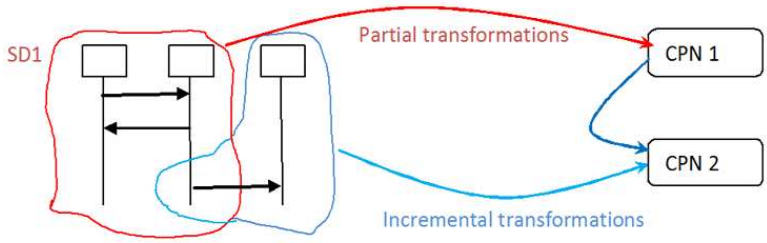


Figure 6.21: An overview of partial and incremental transformation.

As described in Section 3.1.6, the notion of a *region* facilitates to separate a set of interactions using *lifeline decomposition* and *interaction-use* with the *ref* interaction fragment (see Figure 6.22 and Figure 6.23).

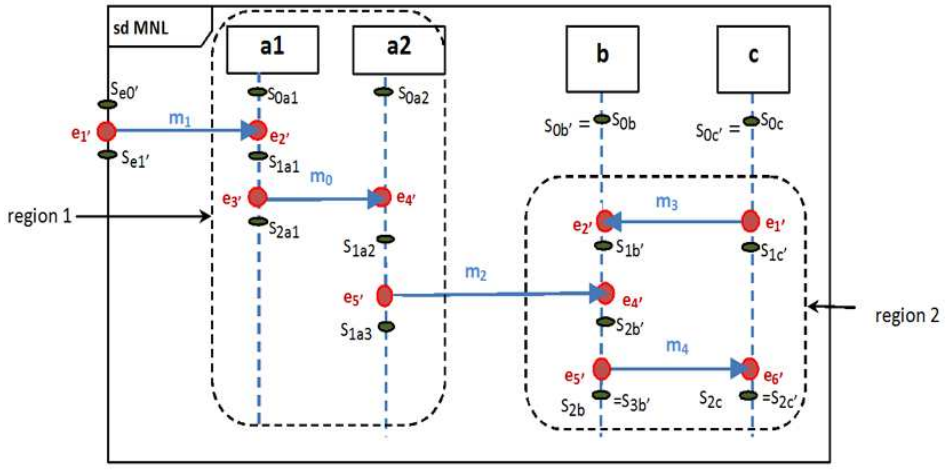


Figure 6.22: A sequence diagram with regions.

By replacing the regions with decomposition, a separate SD can be obtained for a set of sub-interactions given by the region. Additionally, the synthesised region model can be reused when generating a SD for entire behaviour.

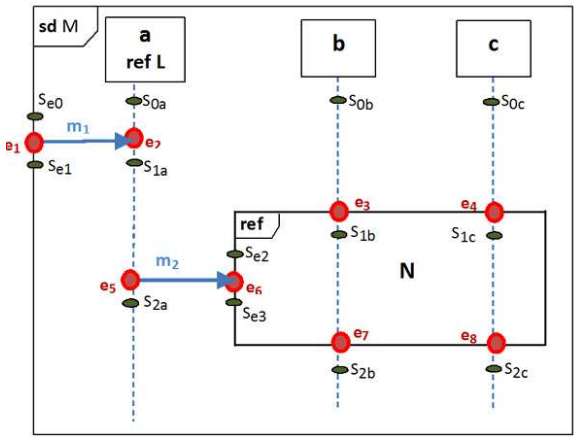
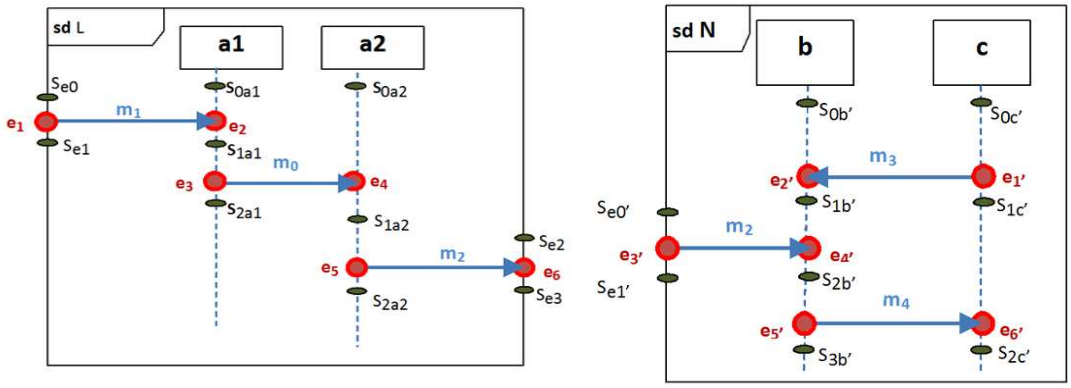


Figure 6.23: A sequence diagram with reference behaviour.



(a) (b)

Figure 6.24: The referred SDs with lifeline decomposition (a) and interaction-use (b) in Figure 6.23.

Consider SD_{MNL} shown in Figure 6.22, where the regions are shown explicitly using a dashed-line enclosing the set of sub interactions. The interactions in each region with the associated events and underlying instances can be separated into SDs on their own. For example, consider SD_L shown in Figure 6.24(a) and SD_N shown in Figure 6.24(b). The diagram SD_L corre-

sponds to the interactions enclosed by *region* 1 that isolates the behaviour of instances $a1$ and $a2$, besides the diagram SD_N shows the interactions enclosed by *region* 2 that communicate between the instances b and the instance c.

Further, the behaviour of SD_L and SD_N are referred by SD_M shown in Figure 6.23 using *lifeline decomposition* and *interaction-use*, respectively. Hence, the behaviour given by the composition of the diagrams SD_M, SD_N, and SD_L are similar to the behaviour of SD_{MNL} (Figure 6.22). Consequently, these individual diagrams can be transformed into corresponding CPNs for analysis separately, thus facilitates the partial analysis of the sub interactions.

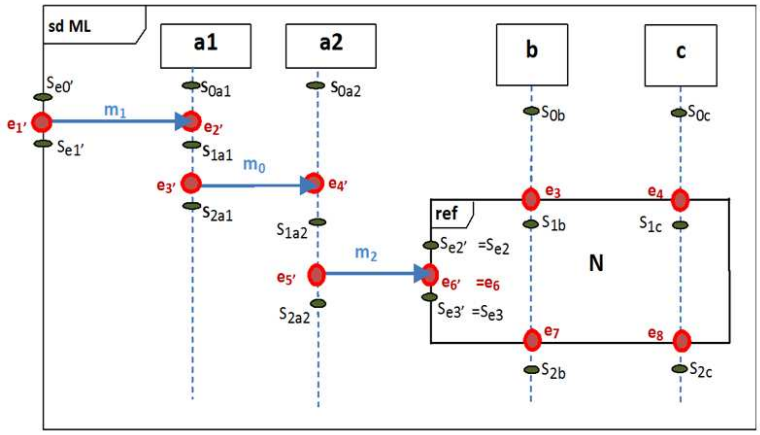


Figure 6.25: A sequence diagram with an interaction-use (ref fragment).

In order to describe the incremental transformations consider the diagrams SD_{ML} in Figure 6.25 that refers SD_N using a ref fragment. The CPN representation for the referred behaviour, CPN_N is shown in Figure 6.26. Here, the diagram SD_{MNL} can be obtained by the composition $SD_{ML} \otimes SD_N$ and Figure 6.27 shows the corresponding CPN_{MNL}.

Initially, CPN_N (Figure 6.26) can be obtained by transforming the behaviour given by SD_N (Figure 6.24(b)) : $CPN_N = \tau(SD_N)$. Then, CPN_{MNL} (Figure 6.27) can be obtained incrementally by combining the previously ob-

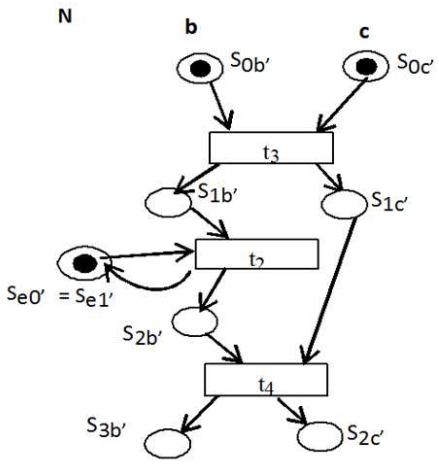


Figure 6.26: The corresponding CPN for the behaviour referred by the fragment ref in SD_{ML}.

tained CPN_N with SD_{ML} as follows:

CPN_{MNL} is obtained by imposing an equality between the places correspond to the state locations $S_{0b}, S_{2b} \in S_{b_{ML}}$ with the places $S_{0b'}, S_{3b'} \in P_N$ in model with the referred behaviour, for the colour b.

Additionally, when there is an environment instance in the SD and a corresponding environment colour in the source CPN, the target CPN ignores the places involved with that colour environment and imposes equality between the common net transitions.

For example, consider the local transition $t_2 = (e_5^{'}, m_2, e_6^{'}) \in T_{ML}$ and the corresponding net transition $t_2^{'} \in T_{n_N}$. The gate event $e_{6'}$ involves with the instance $v \in I^+$ in SD_{ML} and the corresponding colour $v \in \Sigma$ in CPN_N. Here, the target model CPN_{MNL} does not contain a corresponding environment colour $v \in Env$ that is common to both source models. Hence, it ignores the places correspond to $S_{e2}, S_{e3} \in S_{v_{(ML)}}$ and $S_{e0'}, S_{e1'} \in P_N$ and imposes an equality between the corresponding common net transitions: $t_2 = t_2^{'}$ for $t_2^{'}, t_2 \in T_{n_{MNL}}$. Thus shows the incremental transformation. Further, the

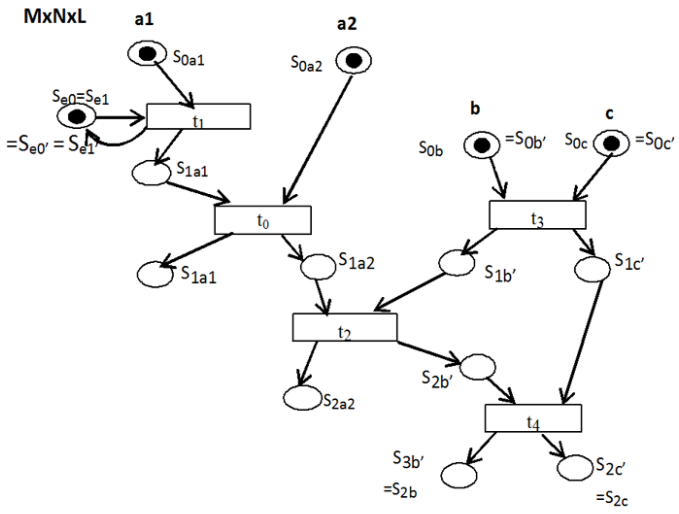


Figure 6.27: The corresponding CPN for $SD_{ML} \otimes SD_N$.

individual model CPN_N can be reused in another model with the reference behaviour.

6.3 Parametric Transformation

Certain software systems with real-time and stochastic behaviours require to model and verify quantitative temporal constraints over interactions [Bondavalli et al., 2005, Kwiatkowska et al., 2007]. Such constraints may include the specification of deadlines, durations, response times, delays, etc., and are represented using timed and stochastic data [Aburub et al., 2007]. The modelling constructs of a single language may be not sufficient to describe the entire requirements of a specific application domain. This may require a modelling language to customise and adapt semantically for the extensions to incorporate the new language constructs. In this context, the term *semantic variability* describes different views of a system using many variants of a given model and enables diverse solutions [Cengarle et al., 2009, Gronniger and Rumpe,

2011, Barbier and Cariou, 2008].

Transformation with parameters can be used to improve new functionalities (values, properties, operations) or to change the application behaviour (activities) or to extend a transformation with new variants, with minimally invasive changes to the existing transformation rules. Hence, supports for model reuse, interoperability, adaptability and management of context information [Vale and Hammoudi, 2008, Kavimandan and Gray, 2011, Vale and Hammoudi, 2009, Mens et al., 2005]. The extensibility and reusability features in Parametric transformation enable to define model transformation rules with minimum effort and less overhead, hence support to increase the modelling power and the software quality as well [Kavimandan and Gray, 2011, Kavimandan and Gokhale, 2007].

We use parametric transformation to reflect different concerns in individual models and to apply model transformation based on parameters (time data, stochastic data, etc.). I.e. parametric transformation extracts only the relevant data from a model that need for a specific transformation. Therefore a single source model can be mapped to multiple target models, each representing a specific concern in the system begin transformed [Vale and Hammoudi, 2008, Kavimandan and Gokhale, 2007]. This supports to explore the semantic variability in the target model for different forms of flexible formal analysis of complex systems.

In particular, we assume a language for specifying interactions, which can capture timeliness, performance and stochastic properties of systems. Such properties can be captured using SD extensions from real-time UML as in [Douglass, 2004], using annotations provided by the UML profile for modelling and analysis of real-time embedded systems (MARTE) [OMG, 2011b], or appropriate extensions of OCL constraints. As defined in Chapter 3: Section 3.1.9

these annotations can be kept separate from the design model, and be passed by a parameter associated with the transformation. In this context, parametric transformation maps a SD to a CPN with different extensions, timed CPNs (TCPNs) and stochastic CPNs (SCPNs). Here, we follow the terminology *semantic models* for the target CPN models (TCPN and SCPN), as suggested in [Boronat et al., 2009b].

The transformation is parametric on the chosen variant with the core set of rules defining the transformation from SDs to CPNs. Moreover, the flexibility of the parametric transformation lies in the incremental nature of the transformation: given a SD (with stochastic and time annotations) and corresponding untimed CPN, other CPN variants can be generated by incrementally applying the specific variant rules. Here, the previously defined transformation rules can be extended with a given parameter, hence supports for incremental transformations. For example the transformation rule for the mapping of a local-transition to a net transition can be extended with stochastic aspects, when the transition contains stochastic data. Similarly, the transformation of a state location can be extended with time data, to obtain the corresponding place in the CPN with time properties.

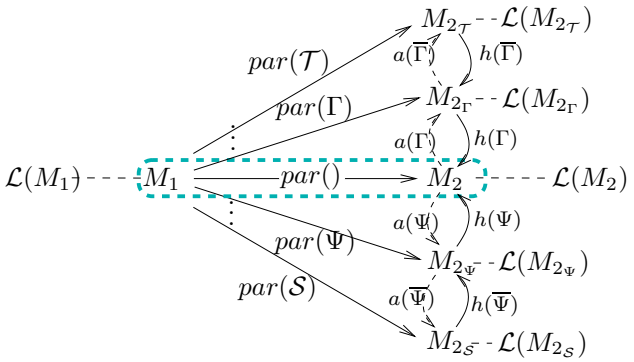


Figure 6.28: The relations between models, variants and languages.

Here, each semantic model variant, TCPNs and SCPNs, is obtained by

passing a parameter in the transformation. It is possible to translate between variants by hiding or adding specific annotations over the software design model. Figure 6.28 illustrates the parametric transformation on a software engineering model M_1 considered in this thesis, with separate sets of timing and stochastic annotations $\mathcal{T}$ and $\mathcal{S}$. The target semantic model is the corresponding CPN variant: CPN for an empty set of annotations, TCPN for timing annotations $\mathcal{T}$, and SCPN for stochastic annotations $\mathcal{S}$. Let $\Gamma \subseteq \mathcal{T}$, and $\Psi \subseteq \mathcal{S}$.

Different transformations can be applied to M_1, for instance $par()$ (denoting a direct parameterless transformation), $par(\mathcal{T})$, $par(\mathcal{S})$, $par(\Gamma)$ or $par(\Psi)$ with $\Gamma \subseteq \mathcal{T}$ and $\Psi \subseteq \mathcal{S}$. With these transformations, the corresponding semantic model variant M_2, $M_{2_\mathcal{T}}$, $M_{2_\mathcal{S}}$, M_{2_Γ} and M_{2_Ψ} can be obtained. Switching between variants is done through transformations a and h, *adding* or *hiding* annotations (not all cases depicted). The flexibility of the approach lies in the fact that we can analyse the effect of certain annotations on the model and change these parameters by adding or hiding. The parametric transformations are partial if some of the details (annotations) of M_1 have to be ignored in the synthesised semantic model.

6.3.1 Transformation of Timed Aspects

As indicated by Ameedeen et al. [Ameedeen et al., 2011], software design models with the notion of timed data enable the analysis of real-time properties and performance of a system. As discussed in Chapter 3: Section 3.1.9, UML sequence diagrams can be extended with a notion of timing aspects to indicate the start time, the time taken by an interaction, or the time interval between two consecutive event occurrences. Similarly, Chapter 4: Section 4.3, has defined the notion of a timed coloured Petri net (TCPN), which is an extension

of a CPN models with the timed aspects.

Timing constraints are usually given by a number to indicate a *fixed delay* or time intervals (with upper and lower bounds) to indicate an *interval delay*. Examples of possible notation include $\{n\}$ for a fixed delay of n time units, and $\{n_1..n_2\}$ for an interval delay between n_1 and n_2 time units, where $n, n_1, n_2 \in \mathbb{R}_0^+$.

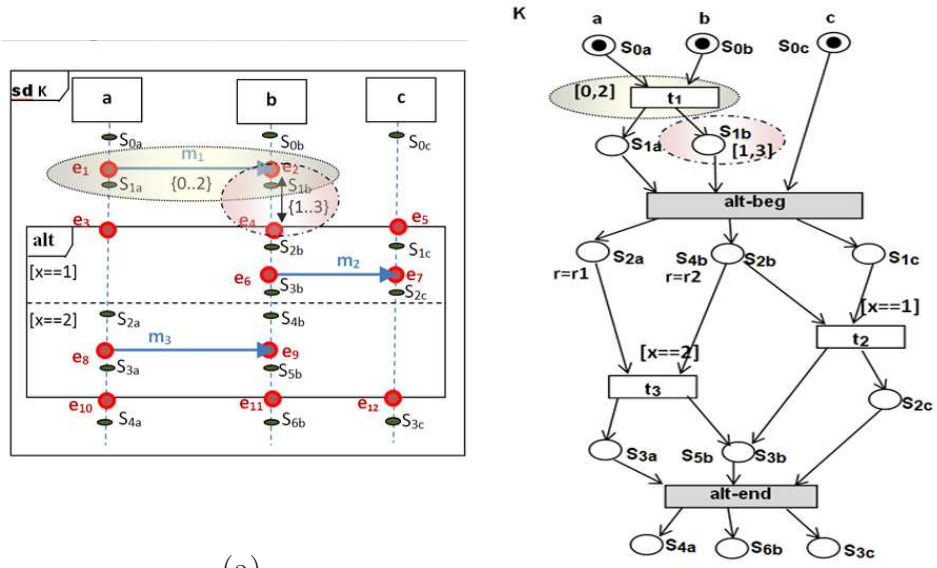
Recalling Definition 3.13, in a SD the timing constraints bound the occurrence of (pairs of) events: $time_{SD} : E \times E \rightarrow \mathbb{R}_0^+ \times \mathbb{R}_0^+$. I.e. time data are represented between events from different lifelines if the events are associated with a local transition, or between two consecutive events on the same lifeline.

The timed data in a SD are mapped to a corresponding TCPN by representing the timing constraints as parameters associated with places and net transitions (see Figure 6.29). These constraints are assigned using a partial labelling function: $time_{CPN} : P \cup T_n \rightarrow \mathbb{R}_0^+ \times \mathbb{R}_0^+$ (Definition 4.5). These time parameters are used to specify the delays on each component.

The timed annotations in a SD can be passed to the target semantic model TCPN by a parameter on the transformation *par* in Figure 6.28. I.e. TCPN for timing annotations $\mathcal{T}$ can be obtained as a variant of a CPN that derived using an empty set of annotations: $par()$.

Consider a time annotation shown in Figure 6.29, $\tau = (t, time_{SD}(e_1, e_2))$ that specifies a constraint on the duration of an interaction that bounds the occurrence of the corresponding send and receive events of the local transition $t \in T$ such that, $time_{SD}(e_1, e_2) = \mathbb{R}_0^+ \times \mathbb{R}_0^+$ where $t = (e_1, m, e_2) \in T$. The corresponding net transition $t \in T_n$ in the TCPN contains timed data with the mapping $time_{CPN}(t) = time_{SD}(e_1, e_2)$. This is considered as the time taken to fire that transition.

When there is a time constraint: $\tau = (e_1, e_2, time_{SD}(e_1, e_2))$ along a lifeline



(a) (b)

Figure 6.29: A sequence diagram with timed data (a) and the corresponding TCPN (b).

between two consecutive events such that $time_{SD}(e_1, e_2) = \mathbb{R}_0^+ \times \mathbb{R}_0^+$ with $e_1 < e_2 \in E_i$ for some $i \in I^+$, it is taken as the time spend on the state location $s \in S$ as given by the *next* function with the first event occurrence: $next_i(e_1) = s$. In the TCPN representation, the timed data is associated with the corresponding place $s \in P$ such that $time_{CPN}(s) = time_{SD}(e_1, e_2)$. This time is considered as the waiting time in that place.

The following Rule 6.9 defines the transformation of timed data in a SD to a TCPN. In the following, let SD_d be a sequence diagram, $\mathcal{T}$ a set of timing annotations over SD_d and let $\Gamma \subseteq \mathcal{T}$.

Rule 6.9 (Timing Annotations) *The model $TCPN_{d,\Gamma}$ obtained by transformation par(Γ) from SD_d is such that for any $\tau = (t, time_{SD_d}(e_1, e_2)) \in \Gamma$ where $t = (e_1, m, e_2) \in T$, $TCPN_{d,\Gamma}$ contains $t \in T_n$ with $time_{CPN_d}(t) = time_{SD_d}(e_1, e_2)$, and for $\tau = (e_1, e_2, time_{SD_d}(e_1, e_2)) \in \Gamma$ with $e_1 < e_2 \in E_i$*

for some $i \in I^+$, $TCPN_{d,\Gamma}$ *is such that* $time_{CPN_d}(s) = time_{SD_d}(e_1, e_2)$ *where* $next_i(e_1) = s$.

The diagram SD_K and the corresponding $TCPN_{K,\mathcal{T}}$ in Figure 6.29 illustrate the timing constraints $\mathcal{T}$ associated with the interactions. The SD contains two timing annotations: on the duration of a local transition and on (consecutive) events along the lifeline $b \in I$. This is given by the set $\mathcal{T} = \{(t_1, [0, 2]), (e_2, e_4, [1, 3])\}$, respectively.

This can be described as $\tau_1 = (t_1, [0, 2]) : [0, 2] = time_{SD_K}(e_1, e_2)$ and $\tau_2 = (e_2, e_4, [1, 3]) : [1, 3] = time_{SD_K}(e_2, e_4)$, respectively, where $\tau_1, \tau_2 \in \mathcal{T}$. Also, the associated state location is $S_{1b} \in S$. By applying Rule 6.9 the corresponding net transition $t_1 \in T_n$ and the place $S_{1b} \in P$ in TCPN are mapped with the timed data such that, $time_{CPN_K}(t_1) = time_{SD_d}(e_2, e_4)$, and $time_{CPN_K}(S_{1b}) = time_{SD_K}(e_2, e_4)$. I.e., the semantic model $TCPN_{K,\mathcal{T}}$ is obtained by transformation $par(\mathcal{T})$.

6.3.2 Transformation of Stochastic Aspects

Software design models with the stochastic annotations facilitate to measure system properties such as performance and mobility [Merseguer and Campos, 2004]. Stochastic data can be represented in sequence diagrams and CPN as described in Chapter 3: Section 3.1.9 and Chapter 4: Section 4.3, respectively. Generally, stochastic behaviour of a system is represented by associating a rate to an interaction (local transition). Here, the rate information corresponds to the movement of an object between two instances.

Recalling Definition **??**, in a sequence diagram a rate value pair annotated on a local transition represents rate associated with the sending and receiving events, respectively. This is given by the labelling function such that $rate_{SD} : T \to \mathbb{R}^+_\top \times \mathbb{R}^+_\top$, where the value is a positive real number (determining

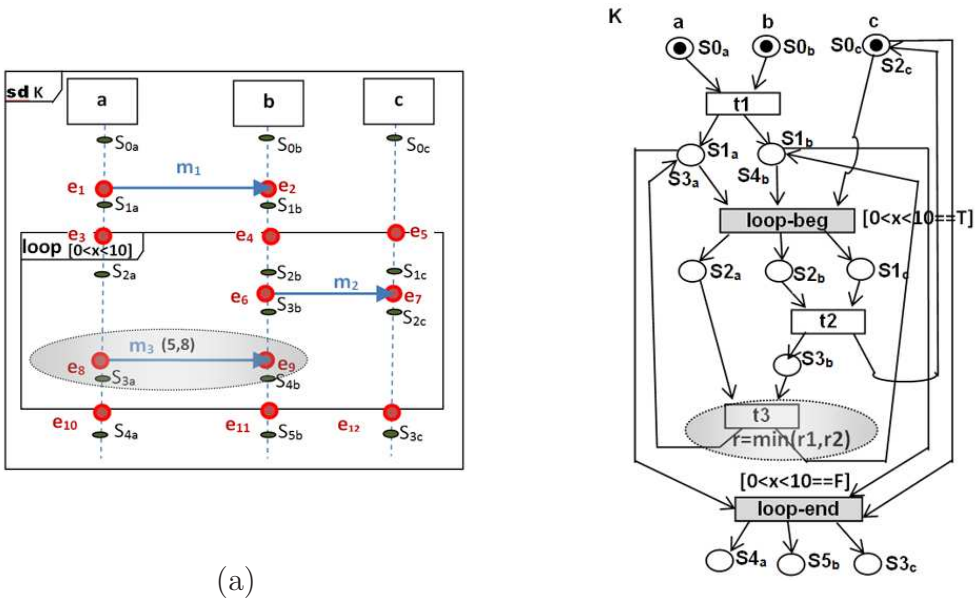
the negative exponential distribution) or an unspecified value (distinguished by the symbol $\top$). The set of positive real numbers together with the unspecified values is specified as $\mathbb{R}^+_\top$. Similarly, as given by Definition 4.6, a TCPN has a partial labelling function on a net transition to indicate the rate value associates with it such that $rate_{CPN} : T_n \rightarrow \mathbb{R}^+$.

The transformation of stochastic data from a SD to the target semantic model SCPN, which is a variant of a CPN, can be obtained by the transformation $par(\mathcal{S})$ in Figure 6.28. The parameter $\mathcal{S}$ represents the stochastic annotations.

When mapping the rate value pair associated with a local transition in a SD to the SCPN, the corresponding net transition is assigned with a synchronised rate that is determined by the minimum of two rates. I.e. a stochastic annotation $\mathcal{S} = \{\sigma \mid \sigma = (t, rate_{SD}(t))\}$, for $t \in T$ in a SD is mapped to a SCPN such that $rate_{TCPN}(t) = min(rate_{SD}(t))$ for $t \in T_n$. Rule 6.10 defines this transformation.

Rule 6.10 (Stochastic Annotations) *Let $\mathcal{S}$ indicates the stochastic annotations over SD_d and $\Psi \subseteq \mathcal{S}$. The model $SCPN_{d,\Psi}$ obtained by transformation $par(\Psi)$ from SD_d is such that for any $\sigma = (t, rate_{SD_d}(t)) \in \Psi$ with $rate_{SD_d}(t) = (r_1, r_2)$ then $SCPN_{d,\Psi}$ contains $t \in T_n$ with $rate_{CPN_d}(t) = min(r_1, r_2)$, where $r_1, r_2 \in \mathbb{R}^+_\top$.*

Consider SD_K and the semantic model $SCPN_{K,\mathcal{S}}$, shown in Fig. 6.30. The SCPN is obtained from the SD using the parametric transformation $par(\mathcal{S})$ where the stochastic annotation over SD_K is given by $\mathcal{S} = \{(t_3, (5, 8))\}$. I.e. the local transition $t_3 \in T$ is associated with stochastic data, where the value 5 and 8 represent the rate associated with the sender and the receiver, respectively such that $rate_{SD_K}(t_3) = (5, 8)$. Here, by applying Rule 6.10 the corresponding net transition $t_3 \in T_n$ in the $SCPN_{K,\mathcal{S}}$ is assigned with the



(a) (b)

Figure 6.30: A sequence diagram with stochastic data (a) and the corresponding SCPN (b).

stochastic data such that $rate_{SCPN_K}(t_3) = min(5, 8)$. I.e. the net transition t_3 fires with a rate of value 5.

Similarly, additional rules can be derived from the above to allow the transformations $a()$ and $h()$ from Fig. 6.28 for adding and removing annotations as desired.

6.4 Hierarchical Transformations

Hierarchical modelling supports for modelling and analysis of the large-scale software systems by visualising different views in different levels of details and enabling model reuse [Baresi et al., 2011, Elkoutbi and Keller, 1998]. This section shows the possibility of transforming different design models with high-level views into sequence diagrams and CPNs, thus supports the underlying model transformation framework from SDs to CPNs.

6.4.1 IOD to Sequence Diagram Transformation

Interaction overview diagrams (IODs) provide an overview for the control flow of a software system design [OMG, 2011a]. By representing the control flow in a hierarchical view, IOD supports for reducing the complexity of a large scale SDs and gives a clear structural understanding among a set of SDs [Kloul and Kuster-Filipe, 2005]. With this high-level design model, IOD supports for partial and incremental transformations to SDs, hence enable partial analysis of a system model.

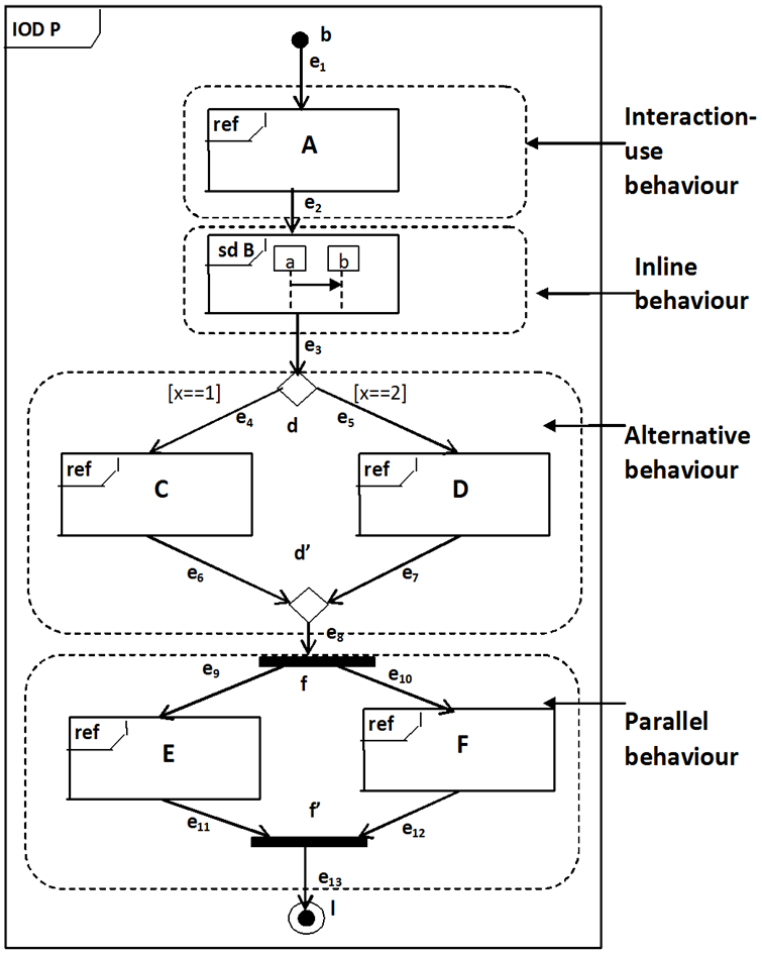


Figure 6.31: The control behaviours of an IOD.

As described in Chapter 3: Section 3.2.2, the activity nodes (*inline-interactions* or *interaction-uses*) in an IOD represent the reference behaviours in a SD. The control nodes that represent the alternative behaviour (*decision, merge*) and the parallel behaviour (*fork, join*) can be transformed into the *alt* and *par* interaction fragments in a SD representation.

Further, these control nodes are properly nested indicating the beginning and the end of an interaction fragment. For example, consider IOD_P shown in Figure 6.31. Here, the activity nodes $r_A, r_C, r_D, r_E, r_F \in R$ indicate the *interaction-use* behaviour and $s_B \in S$ represents the *inline-interaction behaviour*. The control nodes $d \in D_{beg}$, $d^{'} \in D_{end}$ represent the alternative behaviour and the nodes $f \in F_{beg}$, $f^{'} \in F_{end}$ represent the parallel behaviour. Thus an IOD can be transformed to a behaviourally equivalent sequence diagram, facilitating the underlying model transformation and analysis framework.

By recalling the formal representation of the sequence diagram $SD_d = (d, I, E, <, M, T, F, ref, X, Exp)$ (Definition 3.1) and the IOD $I = (N, E, t, l, Exp)$ (Definition 3.1), this sub section defines the transformation of the syntax and semantics of an IOD to a SD. In the following let IOD_d be an interaction overview diagram and the associated SD_d be a sequence diagram named d with a set of state locations S. The transformation rules from an IOD to a SD are as follows.

The name of an IOD is mapped to the name of the corresponding SD.

Rule 6.11 (Name) *Let $d \in \mathcal{N}$ be the name of IOD_d, then the corresponding $SD_{d'}$ contains the same name such that $d = d^{'}$.*

When transforming the basic elements of an IOD, the corresponding SD contains the union of instances given by the activity nodes of the IOD. Here, the initial node and the associated edge are mapped to the initial state locations

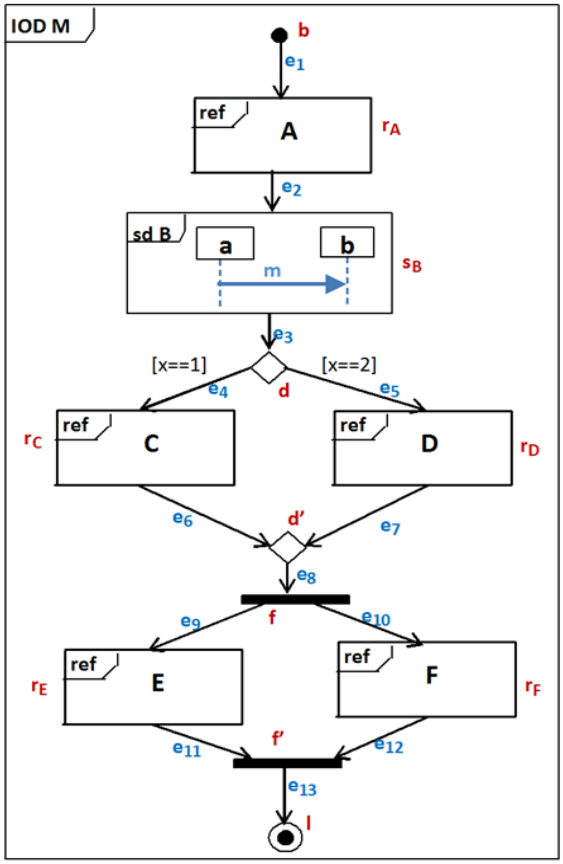
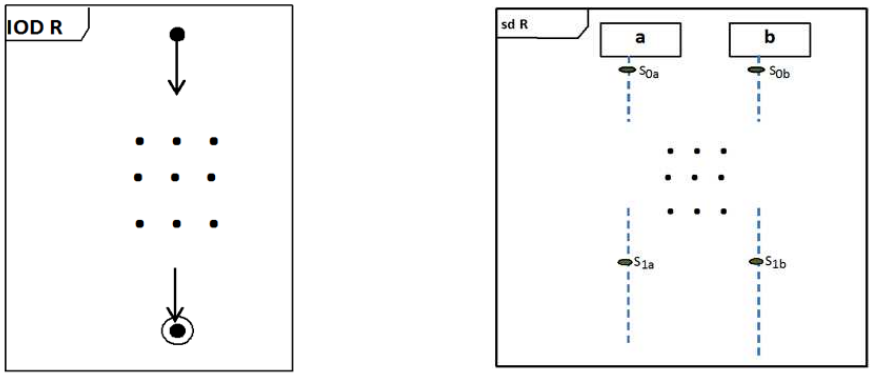


Figure 6.32: A graphical Representation of an IOD.

of the SD. Similarly, the final node and the associated edge are mapped to the end state locations.

Rule 6.12 (IOD initial and final nodes) *Let IOD_a be an interaction overview diagram and let $b \in B$ and $l \in L$ be the initial and the final node. The associated edges $e_1, e_2 \in E$ are such that $next(b) = e_1$ and $next(e_2) = l$. The corresponding SD_a contains instances $i \in I$ such that $j(a) = \bigcup_d j(d)$ where $d \in \mathcal{N}$ for $n \in N_{act}$ and $l(n) = d$. For each initial node b and the associated edge e_1 in the IOD there is a corresponding initial state location $s_i \in S_{ini}$ in the SD. Similarly, for each final node l and the associated edge e_2, there is a corresponding end state locations $s_i^{'} \in S_{end}$ in the SD.*



(a) (b)

Figure 6.33: An IOD representation with initial and final nodes (a) and the corresponding SD (b).

For example, consider IOD_R and the corresponding SD_R shown in Figure 6.33. Let the activity nodes in the IOD implicitly contains instances $a, b \in I$ and the corresponding SD contains instances a and b. The initial node $b \in B$ and the associated edge $e_1 \in E$ are mapped to the initial state locations and the final node $l \in L$ and the associated edge $e_2 \in E$ are mapped to the end state locations, such that $(b, e_1) = \{S_{0i}\}$ and $(l, e_2) = \{S_{1i}\}$ where $S_{0i} \in S_{ini}$ and $S_{1i} \in S_{end}$ for $i = \{a, b\}$.

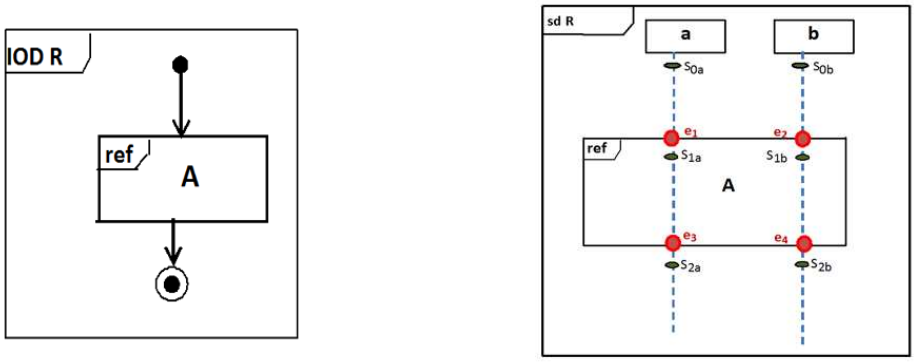
Consider the transformation of an activity node with an *interaction-use* behaviour to a SD. The corresponding SD contains an interaction fragment ref that refers to another SD given by the label of the activity node. Further, the incoming edge associated with the node is mapped to the set of state locations before the ref fragment. Similarly, the outgoing edge associated with the node is mapped to the set of state locations after the ref fragment and preserves the interaction execution order.

The following Rule 6.13 defines the transformation of an activity node with interaction-use to the corresponding SD.

Rule 6.13 (IOD interaction-use behaviour)) *Let IOD_a be an interaction overview diagram and let $r \in R \subseteq N_{act}$ be an activity node that represents interaction-use behaviour such that $l(r) = d$ that refers a sequence diagram SD_d. The associated edges $e_1, e_2 \in E$ are such that $next(e_1) = r$ and $next(r) = e_2$.*

The corresponding SD_a contains $x \in F$ such that $f(x) = (ref, 1)$ where $ref(x) = d$. Let $e, e^{'} \in E_i$ denote the minimal and maximal event in $\overline{g(x)_i}$ for a given instance respectively. The state locations $s, s^{'} \in S_{int}$ are such that $e \in next_i(s)$, $next_i(e_2) = \theta_i(x) = s^{'}$.

For the edges linked with the activity node of the IOD, there are corresponding state locations in the SD such that, $e_1 = \forall_i\{s\}$ and $e_2 = \forall_i\{s^{'}\}$,for $i \in I$.



(a) (b)

Figure 6.34: The interaction-use behaviour of an IOD (a) and the corresponding SD (b).

Figure 6.34 shows an IOD_R with an activity node and the corresponding SD_R. The IOD contains $r_A \in R \subseteq N_{act}$ where $l(r_A) = A$ that refers to another SD_A (Figure 6.35). Let $e_1, e_2 \in E$ be the incoming and outgoing edges associated with the activity node, respectively such that $next(e_1) = r_A$

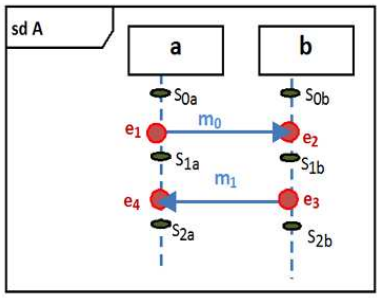
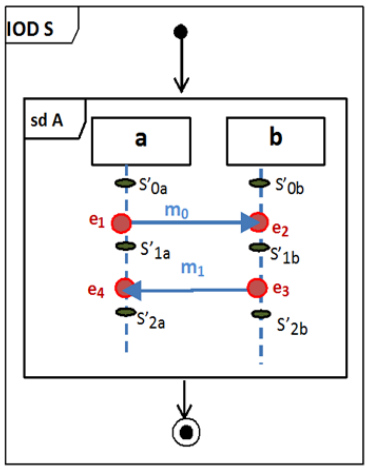
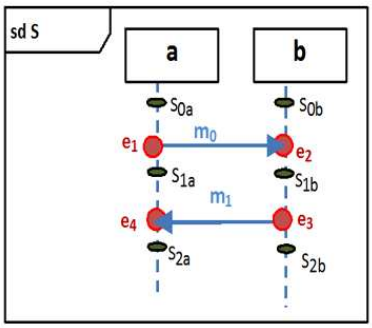


Figure 6.35: The referred sequence diagrams by the IOD in Figure 6.36.

and $next(r_A) = e_2$. . The corresponding SD contains an interaction fragment with reference behaviour: $x \in F$ such that $f(x) = (ref, 1)$ where $ref(x) = A$. Here, the state locations $S_{0a}, S_{0b} \in S_{int}$ before the fragment and the state locations $S_{2a}, S_{2b} \in S_{int}$ after the fragment correspond to the edges e_1 and e_2, respectively.



(a)



(b)

Figure 6.36: The inline behaviour of an IOD (a) and the corresponding SD (b).

When transforming an activity node with *inline behaviour*, the behaviour within the corresponding SD is same as the interactions within the activity

node. Additionally, for each incoming and the outgoing edge lined with the activity node there is a corresponding set of initial and end state locations of the SD, respectively. Rule 6.14 defines this transformation.

Rule 6.14 (IOD inline-interaction behaviour)) *Let IOD_a be an interaction overview diagram and let $s \in S \subseteq N_{act}$ be an inline activity node such that $l(s) = d$ that inline with a sequence diagram SD_d. Let the initial and end state locations of SD_d be $s_1 \in S_{ini}$ and $s_2 \in S_{end}$ for a given instance $i \in I$. The associated edges $e_1, e_2 \in E$ are such that $next(e_1) = s$ and $next(s) = e_2$.*

Here, there is an equality between the SD given by the inline activity node and the target SD : $SD_a = SD_d$. Also, the edges linked with the activity node are mapped to the set of initial and end state locations in SD_a such that $e_1 = \forall_i\{s_1\}$ and $e_2 = \forall_i\{s_2\}$.

Consider IOD_S and SD_S shown in Figure 6.36. The IOD contains an activity node with *inline SD*: $s_A \in S$ such that $l(s_A) = A$ where $A \in \mathcal{N}$. Let $e_1, e_2 \in E$ be the incoming and outgoing edges associated with the node, respectively such that $next(e_1) = s_A$ and $next(s_A) = e_2$.

By applying Rule 6.14, the corresponding SD_S contains the same behaviour as of SD_A. Further, the initial state locations $S_{0a}, S_{0b} \in S_{ini}$ correspond to the edge e_1 and the end state locations $S_{2a}, S_{2b} \in S_{end}$ correspond to the edge e_2.

When the activity nodes are linked in a sequential order, the transformation of the activity nodes are slightly different from Rule 6.13 and Rule 6.14. In this case to preserve the execution order, the corresponding SD contains a *strict* interaction fragment. The number of operands in the fragment is same as the number of consecutive activity nodes and the corresponding transformation of each node k is mapped into the k^{th} operand.

Following Rule 6.15 defines the transformation of a set of consecutive activity nodes in an IOD to a SD.

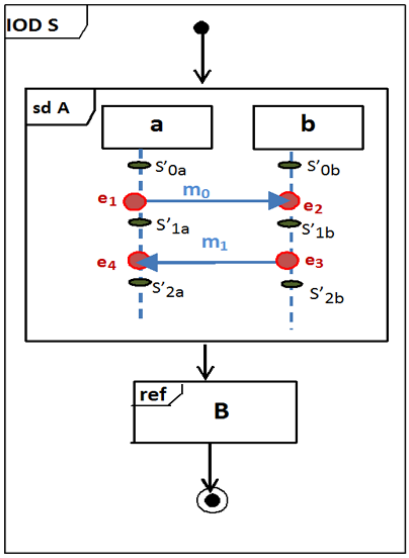


Figure 6.37: The inline behaviour of an IOD.

Rule 6.15 (IOD Sequential behaviour)) *Let IOD_a be an interaction overview diagram and let $n_1, n_2, \cdots, n_n \in N_{act}$ be a set of consecutive activity nodes that refer SDs such that $l(n_1) = d_1$, $\cdots$, $l(n_n) = d_n$. Let $e_1, e_2, \cdots, e_{(n+1)} \in E$ be the associate edges such that $next(e_1) = n_1$, $next(n_1) = e_2$, $next(e_2) = n_2$, $\cdots$, $next(n_n) = e_{(n+1)}$.*

The corresponding SD_a contains a strict interaction fragment $x \in F$, such that $f(x) = (strict, n)$ and let $s_k = min(\lambda_i(x,k))$ and $s_k^{'} = max(\lambda_i(x,k))$ be the minimum and maximum state locations in each operand, respectively, for a given instance and $1 \leq k \leq n$. Further, let $e_{beg}, e_{end} \in E_i$ denote the minimal and maximal event in $\overline{g(x)_i}$ respectively and let $e_{beg} \in next_i(s)$, $next_i(e_{end}) = \theta_i(x) = s^{'}$, for $i \in I$.

Here, each activity node n_k is transformed into the operand k using Rule 6.13 or Rule 6.14, to represent the strict sequencing behaviour,. Here, the minimal and maximum state location sets of an operand correspond with the incoming and outgoing edges that are linked with the corresponding node in the IOD,

respectively, such that $\forall_i\{s_k\} = e_k$ *and* $\forall_i\{s_k^{'}\} = e_{(k+1)}$.

Further to preserve the flow, the first and last edges of the consecutive set are mapped to the state location sets before the beginning and the end of the fragment, respectively, such that $e_1 = \bigcup_i s$ *and* $e_{(n+1)} = \bigcup_i s^{'}$.

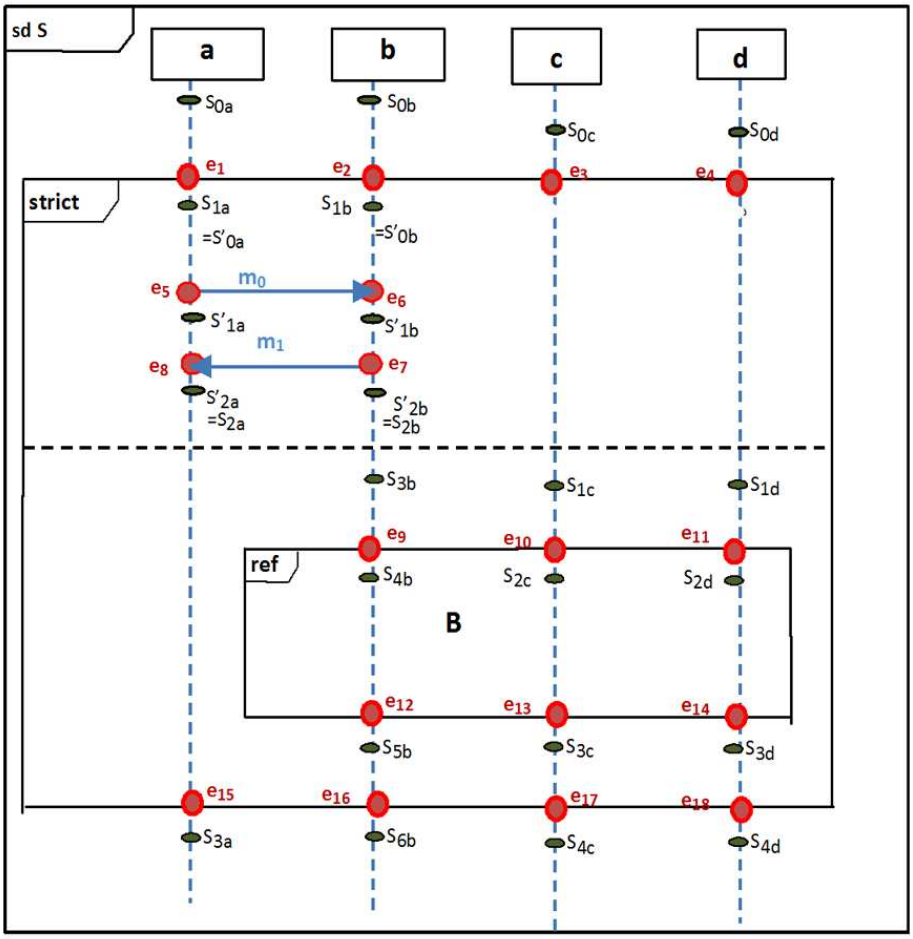


Figure 6.38: The SD for the corresponding IOD with inline behaviours.

For example, consider IOD_S shown in Figure 6.37 and the corresponding SD_S in Figure 6.38. The IOD contains two consecutive activity nodes $s_A, r_B \in N_{act}$ where $s_A \in S$ and $r_B \in R$. Let $e_1, e_2, e_3 \in E$ be the associated edges such that $next(e_1) = s_A$, $next(s_A) = e_2$, $next(e_2) = r_B$ and $next(r_B) = e_3$.

By applying Rule 6.15, SD_S contains a fragment $x \in F$: $f(x) = (strict, 2)$. The minimum and maximum state locations within the each operand are such that $min(\lambda(x,1)) = \{S_{1a}, S_{1b}\}$, $min(\lambda(x,2)) = \{S_{3b}, S_{1c}, S_{1d}\}$, $max(\lambda(x,1)) = \{S_{2a}, S_{2b}\}$, $max(\lambda(x,2)) = \{S_{5b}, S_{3c}, S_{3d}\}$. The k^{th} activity node in the IOD is transformed into the k^{th} operand in the *strict* fragment using Rule 6.13 or Rule 6.14. Additionally, to preserve the execution flow the edges are mapped to the state location sets such that $e_1 = min(\lambda(x,1))$, $e_2 = max(\lambda(x,1)) = min(\lambda(x,2))$ and $e_3 = max(\lambda(x,2))$.

Further, considering the first node s_A and the last node r_B, the associated first and last edges of the consecutive set of nodes are also mapped to the state location sets before and after the *strict* interaction fragment, respectively, such that $e_1 = \{S_{0a}, S_{0b}, S_{0c}, S_{0d}\}$ and $e_3 = \{S_{3a}, S_{6b}, S_{4c}, S_{4d}\}$.

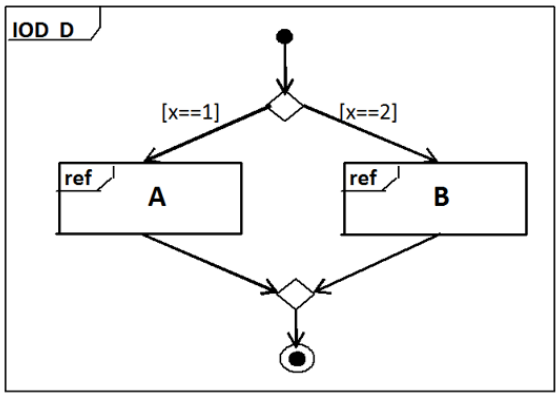


Figure 6.39: An alternative behaviour of an IOD.

Consider the transformation of an alternative behaviour associated with an IOD. For each *(decision, merge)* pair in an IOD, the corresponding SD contains an *alt* interaction fragment (Figure 6.39).

When transforming an IOD with alternative behaviour to a SD, we use an *alt* interaction fragment to represent this behaviour (Figure 6.40). Here, the decision and merge nodes correspond to the beginning and the end of the

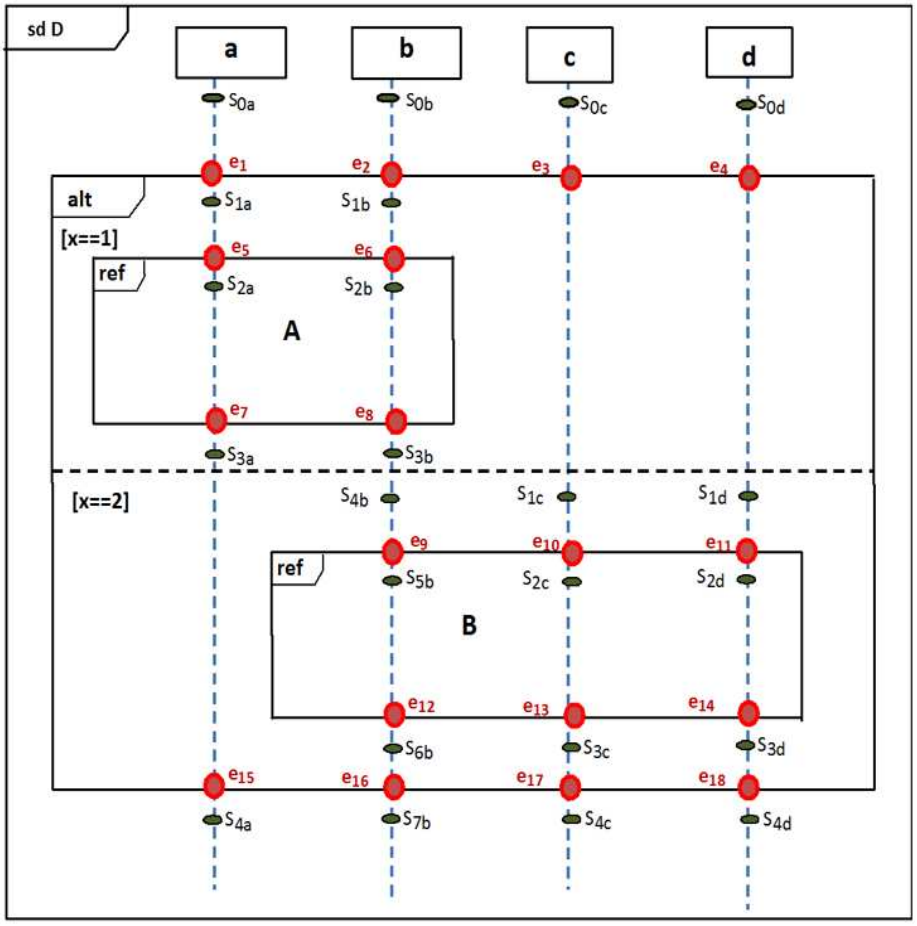


Figure 6.40: The SD for the corresponding IOD with the alternative behaviour.

fragment, respectively. The number of operands in the *alt* fragment are same as the number of outgoing edges linked with the decision node. The constraint associated with each edge of the decision node is mapped to the corresponding operand in the *alt*. Additionally, the behaviours of the activity nodes in each chain within the *(decision, merge)* pair are transformed into the corresponding operand using Rule 6.13 or Rule 6.14 or Rule 6.15.

In order to preserve the flow of control within the diagram, the edges of the IOD are mapped to the set of state locations in the SD as follows. (1)

The incoming edge of the decision node is mapped to the set of state locations before the *alt* fragment. (2) Each outgoing edge of the decision node is mapped to the set of minimum state locations in the corresponding operand. (3) Each incoming edge of the merge node is mapped to the set of maximum state locations in the corresponding operand. (4) The outgoing edge of the merge node is mapped to the set of state locations after the *alt* fragment.

The following rule defines this behaviour.

Rule 6.16 (IOD decision behaviour)) *Let IOD_a be an interaction overview diagram with an alternative behaviour $d \in D$. Let $d_1 \in D_{beg}$ and $d_2 \in D_{end}$ be the decision node and the corresponding merge node, respectively.*

Let the outgoing edges, $c(d_1) = \{e_1, \cdots, e_k, \cdots, e_n\} \in E$, and incoming edges, $c(d_2) = \{e_1^{'}, \cdots, e_k^{'}, \cdots, e_n^{'}\} \in E$, linked with the decision and merge node, respectively such that $|c(d_1)| = |c(d_2)| = n$, where $n \in \mathbb{N}$ is the number of chains within the alternative behaviour.

Further let $e_d, e_d^{'} \in E$ be the incoming and outgoing edges associated with the decision and merge node, respectively, such that $next(e_d) = d_1$ and $next(d_2) = e_d^{'}$. Let $exp_k \in Exp$ be a constraint associated with an edge $e_k \subset c(d_1)$ such that $guard(e_k) = exp_k$ for $1 \leq k \leq n$.

For the alternative behaviour d, the corresponding SD_a contains an alt interaction fragment $x \in F$, such that $f(x) = (alt, n)$ and $exp_k \in Exp$. Further, let $e_{beg}, e_{end} \in E_i$ denote the minimal and maximal event in $\overline{g(x)_i}$ respectively and let $e_{beg} \in next_i(s_i)$, $next_i(e_{end}) = \theta_i(x) = s_i^{'}$. The nodes d_1 and d_2 correspond to the event set $min(\overline{g(x)})$ and $max(\overline{g(x)})$, respectively.

Here, the corresponding constraint exp_k in each chain associates with the first local transition within the operand. Additionally, the behaviour of each chain k within the alternative behaviour is mapped to the corresponding k^{th} operand as expected (Rule 6.13 or Rule 6.14 or Rule 6.15.

In order to preserve the control flow, the minimal and maximum state location sets of an operand correspond with the first and last edge of each chain of the IOD such that $min(g(x,k)) = e_k$ *and* $max(g(x,k)) = e_k^{'}$. *Further, there is a correspondence between the state locations and edges such that* $\forall_i\{s_i\} = e_d$ *and* $\forall_i\{s_i^{'}\} = e_d^{'}$ *for* $i \in I$.

Figure 6.39 shows an IOD with alternative behaviour. IOD_D has an alternative behaviour $d \in D$ with a decision node $d_1 \in D_{beg}$ and the corresponding merge node $d_2 \in D_{end}$. Let the associated edges $e_1, \cdots, e_6 \in E$ are the incoming and outgoing edges associated with the decision and merge nodes such that: $next(e_1) = d_1$, $next(d_1) = \{e_2, e_3\}$, $next(e_4) = next(e_5) = d_2$ and $next(d_2) = e_6$. Also, the edges linked to the nodes can be given by $c(d_1) = \{e_2, e_3\}$ and $c(d_2) = \{e_4, e_5\}$. Thus, the number of chains within the alternative behaviour is $|c(d_1)| = |c(d_2)| = 2$. The constraints associated with each chain can be indicated as $guard(e_2) = [x == 1]$ and $guard(e_3) = [x == 2]$. Additionally the activity nodes associated with each chain are obtained by $r(d,1) = \{r_A\}$ and $r(d,2) = \{r_B\}$ where $r_A, r_B \in N_{act}$.

The corresponding SD_D shown in Figure 6.40 can be obtained by applying Rule 6.16. The SD contains an alternative interaction fragment $x \in F$ where $f(x) = (alt, 2)$. The constraints associated with each alternative chain in the IOD are mapped to the corresponding operand in the fragment. Also, the behaviour within each chain is mapped to the corresponding operand using Rule 6.13 and Rule 6.14.

The edges linked with the nodes d_1 and d_2 correspond to the state location sets such that (1) $e_1 = \{S_{0a}, S_{0b}, S_{0c}, S_{0d}\}$, (2) $e_2 = \{S_{1a}, S_{1b}\}$, (3) $e_3 = \{S_{4b}, S_{1c}, S_{1d}\}$, (4) $e_4 = \{S_{3a}, S_{3b}\}$, (5) $e_5 = \{S_{6b}, S_{3c}, S_{3d}\}$, (6) $e_6 = \{S_{4a}, S_{7b}, S_{4c}, S_{4d}\}$ to preserve the control flow of the interactions.

The parallel behaviour within an IOD is represented using $(fork, join)$

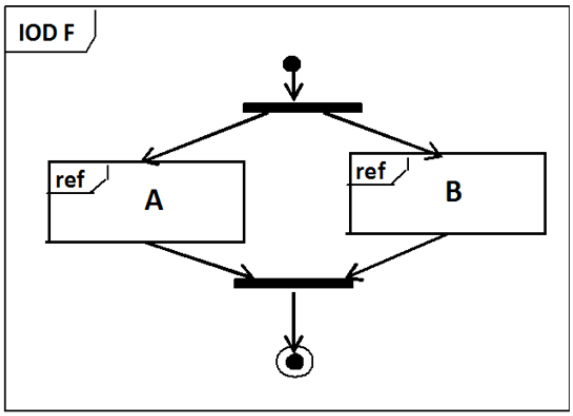


Figure 6.41: A parallel behaviour of an IOD.

nodes pair (Figure 6.41). This behaviour corresponds to the behaviour of a *par* interaction fragment in a SD. Here, the fork and join nodes correspond to the beginning and the end of the fragment, respectively. The number of parallel execution chains are indicated by the cardinality of the outgoing edges linked to the fork node: $|c(f)|$ for $f \in F_{beg}$ and the *par* fragment contains a same number of operands. Additionally, the behaviours of the activity nodes in each chain within the *(fork, join)* pair are transformed into the corresponding operand using Rule 6.13 or Rule 6.14 or Rule 6.15.

In order to preserve the flow of control within the diagram, the edges of the IOD are mapped to the set of state locations in the SD as follows. (1) The incoming edge of the fork node is mapped to the set of state locations before the *par* fragment. (2) Each outgoing edge of the fork node is mapped to the set of minimum state locations in the corresponding operand. (3) Each incoming edge of the join node is mapped to the set of maximum state locations in the corresponding operand. (4) The outgoing edge of the join node is mapped to the set of state locations after the *par* fragment. The following rule defines this behaviour.

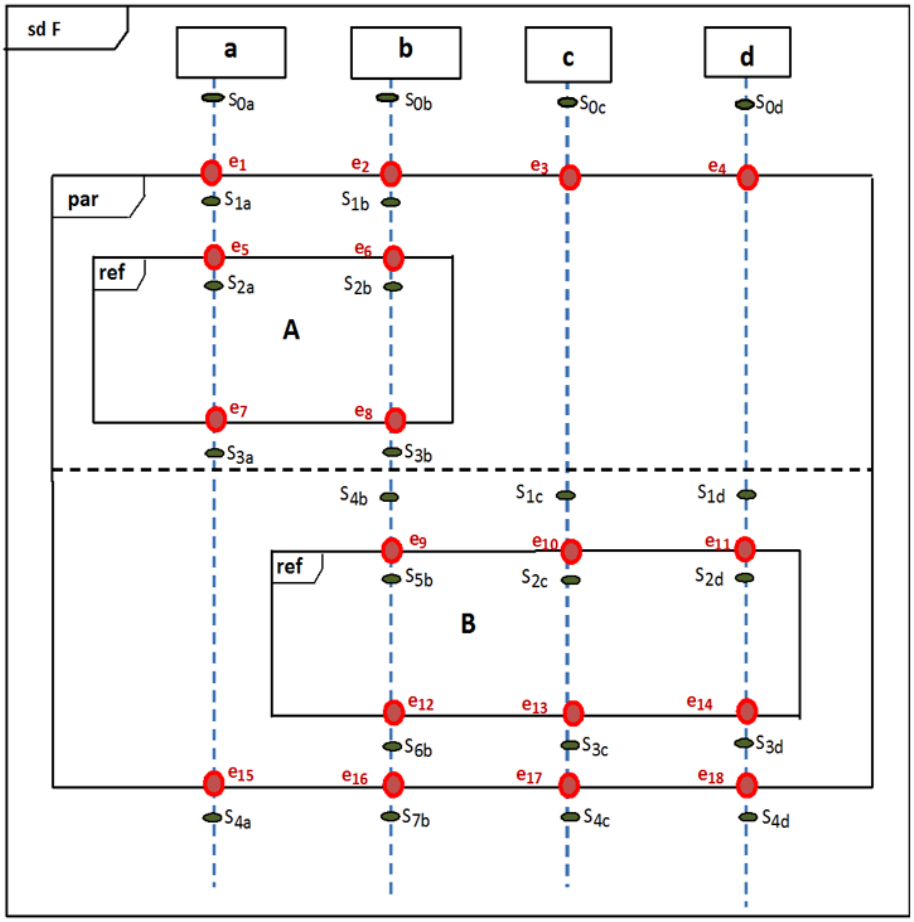


Figure 6.42: A SD for the corresponding IOD with the parallel behaviour.

Rule 6.17 (IOD parallel behaviour)) *Let IOD_a be an interaction overview diagram with a parallel behaviour $f \in F$. Let $f_1 \in F_{beg}$ and $f_2 \in F_{end}$ be the fork node and the corresponding join node, respectively. Let the outgoing edges, $c(f_1) = \{e_1, \cdots, e_k, \cdots, e_n\} \in E$, and incoming edges, $c(f_2) = \{e_1^{'}, \cdots, e_k^{'}, \cdots, e_n^{'}\} \in E$, linked with the decision and merge node, respectively such that $|c(f_1)| = |c(f_2)| = n$ and $1 \leq k \leq n$, where $n \in \mathbb{N}$ is the number of chains within the parallel behaviour. Further let $e_f, e_f^{'} \in E$ be the incoming and outgoing edges associated with the fork and join node, respectively, such*

that $next(e_f) = f_1$ *and* $next(f_2) = e_f^{'}$.

For the parallel behaviour f*, the corresponding* SD_a *contains* $x \in F$ *such that* $f(x) = (par, n)$*. Further, let* $e_{beg}, e_{end} \in E_i$ *denote the minimal and maximal event in* $\overline{g(x)_i}$ *respectively and let* $e_{beg} \in next_i(s_i)$*,* $next_i(e_{end}) = \theta_i(x) = s_i^{'}$*. The nodes* f_1 *and* f_2 *correspond to the event set* $min(\overline{g(x)})$ *and* $max(\overline{g(x)})$*, respectively.*

Additionally, the behaviour of each chain k *within the parallel behaviour is mapped to the corresponding* k^{th} *operand as expected (Rule 6.13 or Rule 6.14 or Rule 6.15). In order to preserve the control flow, the minimal and maximum state locations of an operand correspond with the first and last edge of each chain of the IOD such that* $min(g(x,k)) = e_k$ *and* $max(g(x,k)) = e_k^{'}$*. Further, there is a correspondence between the state locations and edges such that* $\forall_i\{s_i\} = e_f$ *and* $\forall_i\{s_i^{'}\} = e_f^{'}$ *for* $i \in I$*.*

An IOD with parallel behaviour $f \in F$ is shown in Figure 6.41. IOD_F contains a fork node $f_1 \in F_{beg}$ and the corresponding join node $f_2 \in F_{end}$. Let the edges $e_1, \cdots, e_6 \in E$ are the incoming and outgoing edges associated with the fork and join nodes such that: $next(e_1) = f_1$, $next(f_1) = \{e_2, e_3\}$, $next(e_4) = next(e_5) = f_2$ and $next(f_2) = e_6$. Also, the outgoing and incoming edges linked to the nodes can be obtained by $c(f_1) = \{e_2, e_3\}$ and $c(f_2) = \{e_4, e_5\}$, respectively. Thus, the number of chains within the parallel behaviour is $|c(f_1)| = |c(f_2)| = 2$. Additionally the activity nodes associated with each chain are obtained by $r(f,1) = \{r_A\}$ and $r(f,2) = \{r_B\}$ where $r_A, r_B \in N_{act}$.

Figure 6.42 shows the corresponding SD_F for the parallel behaviour of IOD_F. By applying Rule 6.17, the SD contains a parallel interaction fragment $x \in F$ where $f(x) = (par, 2)$. The behaviour within each chain is mapped to the corresponding operand using Rule 6.13 and Rule 6.14. The edges linked with the nodes f_1 and f_2 correspond to the state location sets such that (1)

$e_1 = \{S_{0a}, S_{0b}, S_{0c}, S_{0d}\}$, (2) $e_2 = \{S_{1a}, S_{1b}\}$, (3) $e_3 = \{S_{4b}, S_{1c}, S_{1d}\}$, (4) $e_4 = \{S_{3a}, S_{3b}\}$, (5) $e_5 = \{S_{6b}, S_{3c}, S_{3d}\}$, (6) $e_6 = \{S_{4a}, S_{7b}, S_{4c}, S_{4d}\}$ to preserve the control flow of the interactions.

6.4.2 Sequence Diagram to IOD Transformation

This section shows the possibility of transformations for the behaviour of a SD to an IOD representation. It is important to show a high-level view of a system with many interactions. Here, we consider only the visual transformations, as the detailed transformation from a SD to an IOD can be generated by reversing the IOD-to-SD transformation rules (bidirectional transformations).

A SD can be transformed to an IOD by splitting a SD into mutually exclusive regions, creating separate SDs for each regions and building another SD that refers the newly created SDs using ref fragments. Then the SD with reference behaviour can be transformed into an IOD consist of interaction-use nodes.

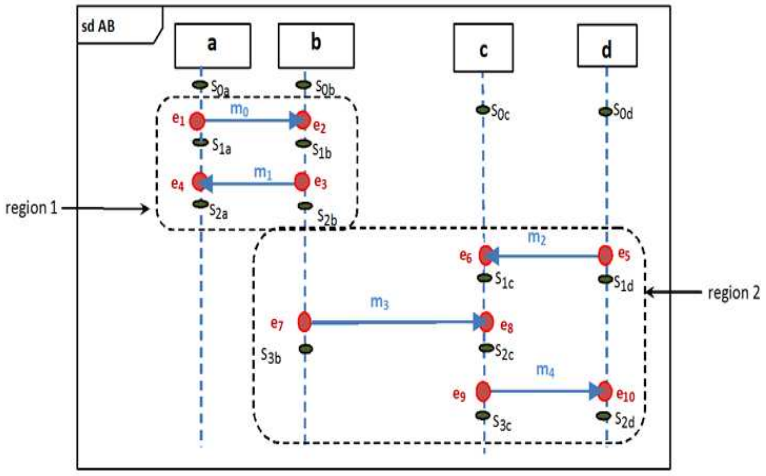


Figure 6.43: A SD with mutually exclusive regions.

The transformation rules defined in Section 6.4.1 can be applied reversely to obtain the transformation from a SD to an IOD. When defining a behaviourally

equivalent IOD from a SD, it is important to satisfy the following two constraints.

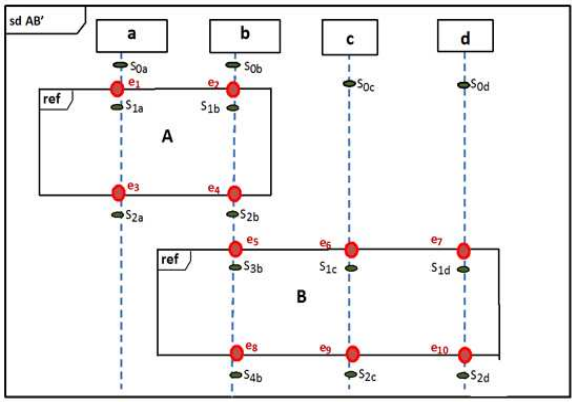


Figure 6.44: The corresponding SD with an interaction-use behaviour.

- The regions of the SD should be mutually exclusive. I.e. the regions should not contain overlapping elements and should be disjoint from each other. Let $r_i \in R$; $i \in \mathbb{N}$ be a set of regions in a SD. Then $r_1 \cap \cdots \cap r_n = \emptyset$ or $\bigcap_{i=1}^{n} r_i = \emptyset$.

- The combination of all regions should form the set of all interactions that belongs to the entire SD. I.e. there should not be interactions within the SD that are not belong to any of the region. Formally, let $r_i \in R$; $i \in \mathbb{N}$ be a set of regions in a SD. Then $r_1 \cup \cdots \cap r_n = \varepsilon$ or $\bigcup_{i=1}^{n} r_i = \varepsilon$, where ε is the universal set that include all the interactions within a sequence diagram.

Consider SD_{AB} shown in Figure 6.43. The interactions are divided into two regions and the reference behaviour is shown in $SD_{AB'}$ in Figure 6.44. These ref interaction fragments are mapped to the $inteacton - use$ nodes in IOD_{AB} shown in Figure 6.45. Thus a behaviourally equivalent IOD can be obtained from a SD representation.

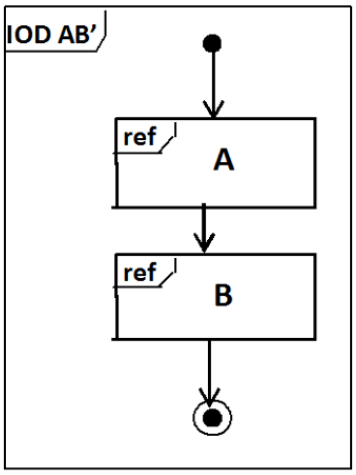


Figure 6.45: An IOD corresponds to SD_{AB} in Figure 6.44

6.4.3 HCPN to CPN Transformation

A Hierarchical coloured Petri net (HCPN) can be easily used to construct an equivalent CPN, and vice versa. In a HCPN a net transition or a colour can be substituted by another CPN, result in multiple layers of details, which brings more details into the model. The net transitions with a reference in a HCPN, can represent the a separate CPN with related colours. Thus, HCPs with high-level abstract representation can be simplified into a CPN that gives a broad view of a system. In this way, the hierarchical models support for managing models of large-scale and complex real-world systems.

A HCPN can always be unfolded into an equivalent non-hierarchical CPN model with the same behaviour. Section 6.1.3 defines the transformation rules for the composition of CPNs considering the hierarchical view of the model using reference behaviours. Here, when a net transition or a colour of a HCPN refers to another CPN (Figure 6.46, the detailed view of the model can be obtained by substituting the referred behaviours. Substitution does not require adding fundamentally new details, it only needs to define and establish the

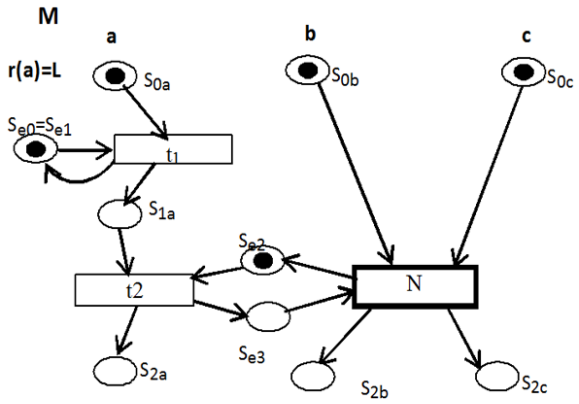


Figure 6.46: A HCPN with a reference behaviour.

proper connections between the relevant places and transitions in both nets. This only changes the graphical structure of the system model, without changing its meaning. Thus, rules defined for *CPNcomp* path in Figure 6.1) can be applied reversely (as a bidirectional transformation) to realise the transformations from a HCPN to a CPN.

6.5 Concluding Remarks

Software engineering models for large-scale systems are usually combinations of models representing different perspectives. This chapter has defined and described more complex mechanisms for generating interaction models (SDs) and their transformations. The defined model composition rules enables to obtain a single model from two or more related models for a unified understanding of the entire system. These rules have been extended for partial and incremental transformations that can apply on a set of sub-interactions in a model and enable partial analysis of the model. Also, this chapter has showed the applicability of incremental transformations by reusing models obtained

using partial transformations.

The parametric transformation defined in this chapter allows the transformation of a SD with time and stochastic annotations to the respective extension of the CPN model, namely TCPN and SCPN, respectively. These transformations explore the semantic variability in the target model to analyse properties in systems with real-time and stochastic behaviour. Thus, applying formal analysis will be feasible by exploring one class of target models with rich variants. Further, this chapter has considered the integration of models considering their hierarchical aspects. These hierarchical modelling and transformation visualise different views in different levels of details and enable reuse of the modelled system.

7 Model Transformation Correctness

Most work on model transformations concentrates on methods and tools for defining and implementing transformations [Whittle and Schumann, 2000, Delatour and Lamotte, 2003,Eichner et al., 2005,Ribeiro and Fernandes, 2006,Fernandes et al., 2007, Ameedeen et al., 2011, Campos and Merseguer, 2006], on identifying classes of transformations of interest [de Lara and Guerra, 2005, Ehrig et al., 2008, Kuster et al., 2004, Mens et al., 2005], and in some cases on proving *confluence* and *termination* properties about transformations [T.Mens and Grop, 2006, Lano, 2009]. *Confluence* holds if from a given source model we are always able to obtain a unique target model, and *termination* indicates that a model transformation always leads to a result, in other words, terminates.

However, little attention is usually given to establishing the *correctness* of a given transformation, i.e., the transformation produces well-formed target models from valid source models (*syntactical* correctness) and preserves the behaviour of the source model (*semantical* correctness).

There is not much research available in establishing semantical correctness of transformations particularly for *exogenous* transformations, that is transformations where the source and target models belong to different classes of models and hence have a different metamodel [Hülsbusch et al., 2010b, Hülsbusch et al., 2010a, T.Mens and Grop, 2006]. Both syntactical and semantical correctness of model transformations are important but technically very different to establish. In the case of transformations of structural models, syntactical correctness is sufficient and a declarative method for specifying model transformations common practice [Cabot et al., 2010c, Hülsbusch et al., 2010b, Orejas et al., 2009, Orejas and Wirsing, 2009, Cabot et al., 2010a, Ehrig and Ermel, 2008, Greenyer and Kindlev, 2007]. By contrast, for transformations of be-

havioural models, semantical correctness is crucial and we have to be able to guarantee that any observable property of the source model is preserved in the target model.

Conversely, a transformation is complete if any observable property of the target model can be traced back to the source model. This is not the case if there are more allowed behaviours in the target model than were expected or specified in the source model. Correctness and completeness for a model transformation, also known as a *strongly consistent* model transformation, essentially means that source and target models are in some sense equivalent by transformation.

In this chapter, we establish that our SD-CPN model transformation is strongly consistent focusing on semantic correctness and completeness. Our main contribution is in Section 7.2. In Section 7.1, we briefly describe the approach generally adopted for (syntactical) correctness of model transformations using graph-based mechanisms. Our previously formally defined transformation rules can all be given in this alternative way and we just show a few examples here. The proof of the semantic correctness is given in steps adding new constructs each time. We reflect on how our proof method can be generalised. We use several examples throughout for illustration.

7.1 Syntactical Correctness

To establish syntactical correctness of model transformations it is common to use a declarative approach. In the declarative approach, visual or textual descriptions of the mappings between source and target models are given. This approach focuses on what needs to be transformed into what by defining a relation between source and target model elements. The representation of each relationship is defined as a *declarative pattern* and based on the metamodels

of the source and target models. From the patterns it is possible to derive operational mechanisms for forward and backward transformation between the models [Orejas et al., 2009, Cabot et al., 2010a]. Further, these metamodel-based declarative rules are complemented with additional information to express relations and constraints between source and target elements.

Syntactical correctness of model transformations is usually based on triple graphs [Schürr, 1995] and graph transformation techniques where models are given by graphs [de Lara and Guerra, 2005]. We show how this can be done for our SD-to-CPN transformation by representing some of our rules using tripple graph grammars (TGGs).

In graph transformation techniques, a transformation rule consists of a source graph (given at the left and referred to as LHS), a target graph (given at the right and referred to as RHS), and a middle graph which establishes the relation. Overall, the model transformation is given by a set of graph transformation rules. When a rule is fired the LHS graph is replaced by the RHS graph. Here, metamodels for the source and target models are used to establish the vocabulary of the LHS and RHS and to ensure that the transformation produces a well-formed target model. Consequently, graph grammar can be applied to any source model conforming to the source metamodel and produce (following the rules of the transformation) a target model conforming to the target metamodel.

Figure 7.1 and Figure 7.2 show the metamodels for the SD and the CPN considered in this thesis, respectively. This section describes only the relevant parts of the metamodel needed for defining transformation rules (shown in Figure 7.3-Figure 7.9) for the syntax of the models. The complete description of the SD and CPN metamodels is given in Chapter 8.

The SD metamodel given in Figure 7.1 is consistent with the elements and

functions defined earlier in the formal representation of SDs given in Chapter 3. Class SD has an attribute *name* and composite associations with classes $Instance$, $Event$, $LocalTransition$, $StateLocation$ and $InteractionFragment$. Class $Instance$ has two subclasses $ObjectInstance$ and $EnvInstance$ that represent object and environment instances respectively. Class $Instance$ has associations to classes $Event$ and $StateLocation$ to represent the events and state locations belonging to an instance respectively. Class $StateLocation$ has three subclasses to indicate initial, internal and end state locations.

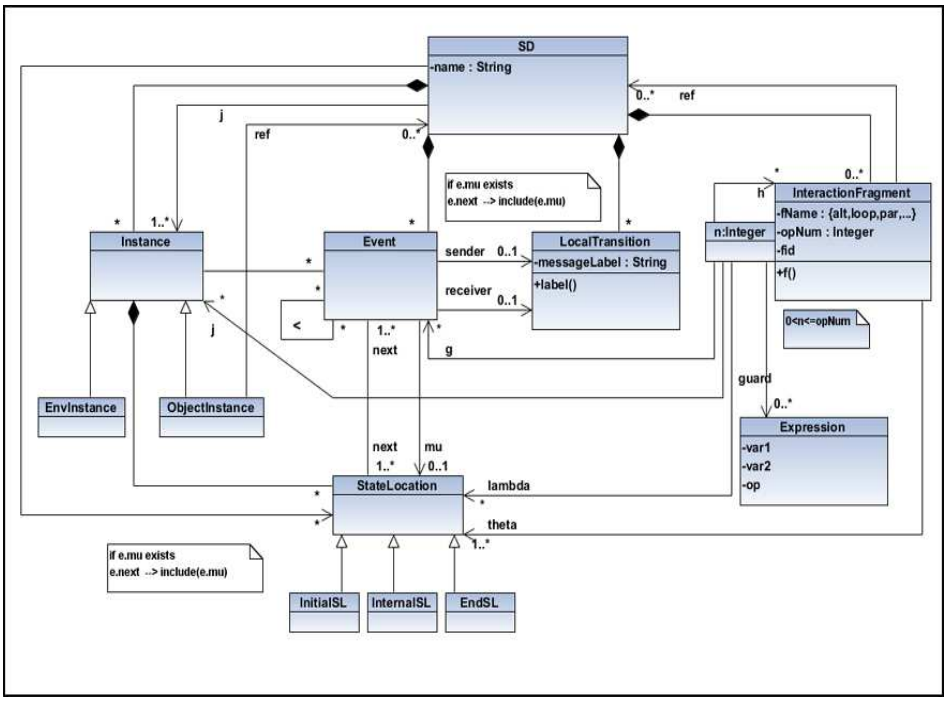


Figure 7.1: SD Metamodel

Class $LocalTransition$ has an attribute *messageLabel* and an operation *label*() that (re)assigns a message label to a local transition. An instance of class $LocalTransition$ has a sender and a receiver event which is given by the two associations (and corresponding rolenames) from $LocalTransition$ to $Event$. Moreover, $Event$ instances are partially ordered, and are associated

with one or more instances of class *StateLocation* (denoting function *next*). An *Event* instance may or may not be associated with a *StateLocation* instance through role *mu* (matching formally defined function μ).

An OCL invariant can be used to indicate that if an event e is involved in a $LocalTransition$ then $e.mu$ is defined and in that case the set of next state locations of event e always includes $e.mu$.

```
context e:Event inv:
if e.LocalTransition<>null then e.next->includes(e.mu)
```

Class $InteractionFragment$ has attributes $fName$, $opNum$ (number of operands), and fid (identifier). A qualifier n (indicating an operand number) is used for associations from this class to $Instance$, $Event$, $StateLocation$ and back to $InteractionFragments$ to denote the formally defined functions j, g, λ and h respectively. A particular operand may also have an $Expression$ that contains variables and an operator. An instance of $InteractionFragment$ may refer to another SD and is given by the association ref to class SD.

Figure 7.2 shows the CPN metamodel that conforms to the CPN Definition 4.1 in Chapter 3. *Place*, $NetTransition$, *Arc*, *Label* and *Colour* classes represent the main constructs and are associated with class CPN using composition. Similarly to the class $Instance$ in the SD metamodel, there are two specialisations of class *Colour*. Class *Place* has attributes *marking* that represents the initial number of tokens associated with the place, and *status* that shows the status of the place (can be complete, safe, etc). Each instance of class *Place* is associated with an instance of class *Colour* (denoting its colour or object type). An instance of class $NetTransition$ may have a link to a CPN as a label, and/or a link with class *Label* (for usual net transitions the latter will be the case). Classes $NetTransition$ and *Arc* may have a guard denoting an instance of *Expression*. The class *Expression* is as defined for the SD

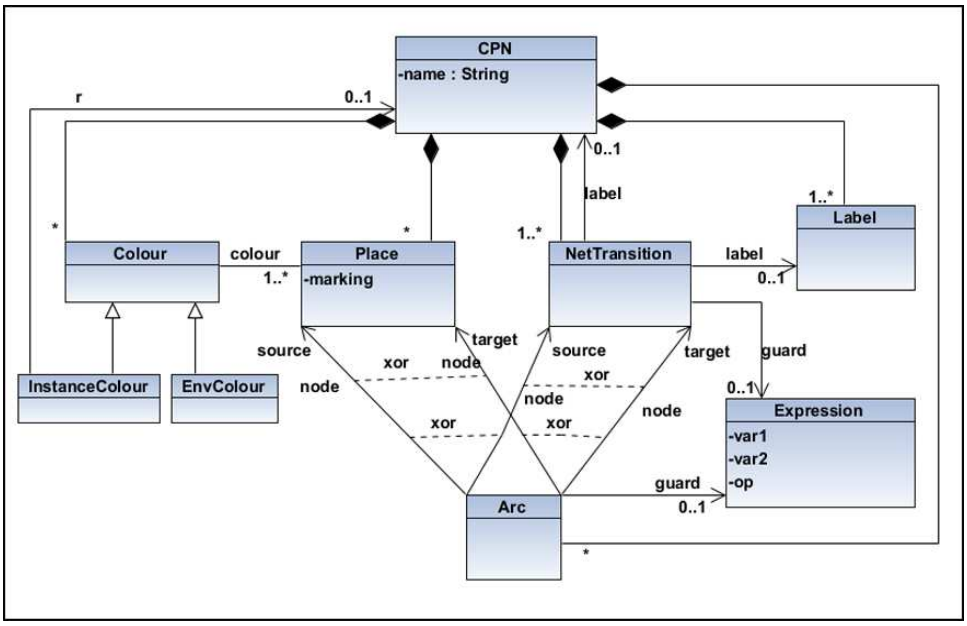


Figure 7.2: CPN Meta-model

metamodel. Finally, instances of class *Arc* are associated with one instance of *Place* and *NetTransition* to reflect the formally defined *node* function. Note that the various constraints `xor` (exclusive or) used in the diagram assure the proper definition of *node* and guarantee, for example, that an arc cannot have a place as both source and target.

Triple graph grammars (TGG) [Schürr, 1995] are a well-known graph transformation approach to define model transformations in a declarative way. The structure of a model is specified by graph grammars. Here, models are defined as pairs of source and target graphs which are connected through an intermediate corresponding graph that embeds into the source and target graphs. Definition 7.1 as given in [Ehrig et al., 2008, Hermann et al., 2010] defines the main constructs of TGGs.

Definition 7.1 (Triple Graph) *A triple graph* $G = (G_S \overset{s_G}{\leftarrow} G_C \overset{t_G}{\rightarrow} G_T)$ *consists of three graphs* G_S*,* G_C*, and* G_T*, called source, correspondence, and*

target graphs, together with two graph morphisms $s_G : G_C \rightarrow G_S$ *and* $t_G : G_C \rightarrow G_T$.

A triple graph as defined has a source G_S, a correspondence G_C and a target G_T graphs, where the intermediate graph establishes the mapping between the other two through the defined morphisms. The TGG rules shown in Figure 7.3-Figure 7.9 describe visually the structural correspondence of the source and target graphs with each other (the implicit morphisms). The rules shown only cover the basic transformation rules described in Chapter 5.

For example, the TGG rules shown in Figure 7.3, Figure 7.4, Figure 7.5, Figure 7.6, Figure 7.7, Figure 7.8, and Figure 7.9 correspond to the SD-to-CPN transformations given by Rule 5.1, Rule 5.2, Rule 5.3, Rule 5.5, Rule 5.6, Rule 5.15, and Rule 5.10, respectively.

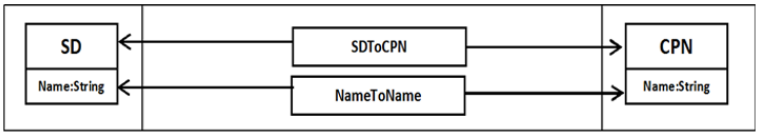


Figure 7.3: TGG rule to transform a name of a SD to the corresponding name of the CPN, and $\tau(SD) = CPN$.

Figure 7.4: TGG rule to transform an instance of a SD to the corresponding colour of the CPN.

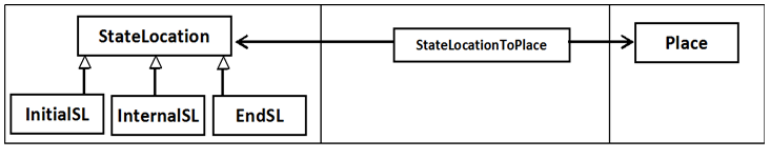


Figure 7.5: TGG rule to transform a state location of a SD to the corresponding place of the CPN.

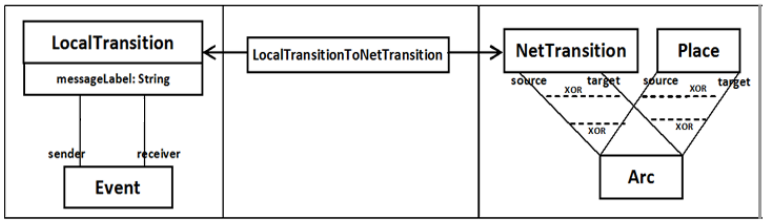


Figure 7.6: TGG rule to transform a local transition of a SD to the corresponding net transition of the CPN.

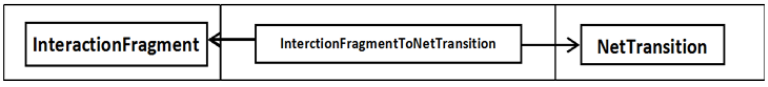


Figure 7.7: TGG rule to transform a message label of a SD to the corresponding label of the CPN.

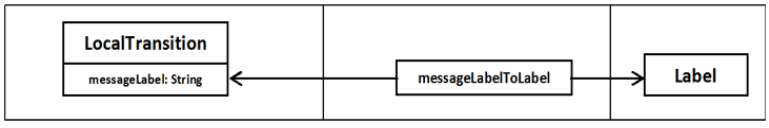


Figure 7.8: TGG rule to transform an interaction fragment of a SD to the corresponding unlabelled net transitions of the CPN.

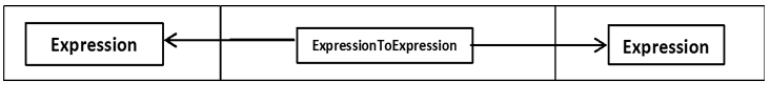


Figure 7.9: TGG rule to transform an expression of a SD to the corresponding expression of the CPN.

TGG transformation rules are structured in three columns, where source, correspondence mapping and target graphs are specified in the left, middle and right columns, respectively. A graph grammar rule is applied by substituting the left-hand side (LHS) with the right-hand side (RHS), if the structure of the LHS can be matched to a graph. These rules are based on the elements of the corresponding metamodels and specify the mapping of the elements of a source model to the corresponding elements of the target model. For example in Figure 7.3, the name of the SD corresponds to the name of the CPN, instances correspond to colours, state locations correspond to places, and so on. Since TGGs use the metamodels of the corresponding models to define the transformations, inconsistencies caused by the transformations can be avoided [Greenyer and Kindlev, 2007]. In actual transformation, the nodes in the correspondence mapping column are instantiated and keep track of the corresponding model structure. For simplicity, we have not shown the objects associated with each instance and the association types between the instances.

From the following theorem and proof given in [Hermann et al., 2010] (proof not reproduced here), we can state the (syntactical) correctness and completeness of each model transformation given by the TGG rules as follows.

Theorem 7.1 (Model Transformation Correctness and Completeness) *Let a model transformation* $MT : M_S \Rightarrow M_T$ *be defined by all model trans-*

formation rules (G_S, G_C, G_T) *where* $G_S \in M_S$ *and* $G_T \in M_T$. *Each model transformation MT is*
- correct, if for each model transformation rule (G_S, G_C, G_T) *there is a valid* $G = (G_S \leftarrow G_C \rightarrow G_T)$, *and it is*
- complete, if for each $G_S \in M_S$ *there is a valid* $G = (G_S \leftarrow G_C \rightarrow G_T)$ *with a model transformation rule* (G_S, G_C, G_T).

A further advantage of using TGGs is that it allows us to represent bidirectional model transformations. The formal theory underlying TGGs can also be used to explore the coverage of the model transformation to check whether some of the source elements are not mapped onto target elements, and conversely whether target elements are not reached by transformation. One problem of using this approach is that it does not show how to apply the transformation.

The TGG-based approach offers an implicit interpretation to formulate the desired specification of a model transformation with a set of underlying mechanisms. Thus a declarative approach is more concise than a comparable operational approach. However, there is a trade-off between conciseness and comprehension. For example, when a transformation has too many implicit and complicated concepts, it may be more difficult to understand than a more explicit, yet verbose, model transformation. Hence, we have used an operational approach to define our model transformation rules formally (defined in Chapter 5 and 6) for the semantic correctness of the mapping between the models and the declarative based model transformation rules to show the syntactical correctness of the model transformations.

7.2 Semantical Correctness

Semantical correctness is an important requirement of model transformations to ensure the preservation of the behaviour of the original model by transformation [Christensen and Petrucci, 2000, Lakos and Petrucci, 2004, Grumberg and Long, 1991]. Semantical correctness of a model transformation shows that the behaviour of the generated target model contains (and ideally is equivalent) to the source model. For instance, when transforming UML models into mathematical models, the results of a formal analysis can be invalidated by erroneous model transformations as it becomes impossible to know whether an error is a consequence of bad design or incorrect transformation. Despite its importance, semantical correctness of model transformations remains hard to prove [Orejas and Wirsing, 2009, Greenyer and Kindlev, 2007].

Previously, we have defined a formal representation for UML 2 sequence diagrams with additional notions of traces and language (set of legal traces) (Chapter 3), a formal representation for CPNs as needed with the notions of traces and language (Chapter 4), and formal transformation rules to obtain a CPN from a given UML 2 SD (Chapter 5 and 6). These model transformation rules are based on an operational model transformation approach that explicitly describes the operations needed to create elements in the target model from elements in the source model [Orejas et al., 2009]. This approach focuses on how and when the transformation is performed by specifying the required steps to derive the target model from the source model [T.Mens and Grop, 2006].

Here, we show that the transformation rules defined guarantee a one-to-one correspondence between the set of legal traces of both models, that is, the languages are *equivalent* also known as *strongly consistent*. In particular, with the strongly consistent nature of our transformation, we are certain that

the synthesised model (CPN) does not contain implied behaviours and can be used for an accurate analysis of the source SD using the existing analysis tools available for CPNs. Our result proves the semantic correctness of the defined model transformation.

The following definitions 7.2-7.5 recall the main notions introduced in chapters 3 and 4. Here, the alphabet L of a sequence diagram SD is defined over the set of messages M and the set of legal traces of a SD determines the language of an SD given by $\mathcal{L}_1(SD)$. Similarly for CPNs the language is given by $\mathcal{L}_2(CPN)$ and uses the same alphabet of labels M.

Definition 7.2 (SD Trace) *A trace of a sequence diagram SD with set of state locations S is a possibly infinite word w, $w = m_1 \cdot m_2 \cdot m_3 \ldots$ over the alphabet L_1 iff there exists a chain c of state locations and events for some instance $i \in I$ such that we can derive w from c.*

The main idea of a chain for a given object instance $i \in I$ (and we do not care about environment instances for generating traces), is that it is an interleaving of state locations and events where each object instance may have (depending on the interactions it is involved in) more than one chain. A chain starts at an initial state location (for the object at hand i) and using $next_i$ we obtain all the following events and state locations. Every time $next_i$ returns a set of two or more elements rather than a singleton we have a *branch*. For a particular chain, for instance, $s_1 \cdot e_1 \cdot s_2 \cdot e_2 \ldots$, a trace is derived in such a way that only events e_n in the chain involved in a local transition (i.e., $(e_n, m, e) \in T$ or $(e, m_n, e_n) \in T$) are considered and give raise to the trace $m_1 \cdot m_2 \ldots$. In other words, if an event denotes the beginning/end of an interaction fragment they are ignored and do not provide useful information for the trace. In particular this also means that different chains (derived from

different SDs for example) can have the same underlying trace. This is related to a notion of bisimulation of SDs to which we will return later on.

Definition 7.3 (SD Language) *The language for an SD is the set* $\mathcal{L}_1(SD)$ *of words over the alphabet* L_1*, where* $\mathcal{L}_1(SD) = \{W|$ *W is a maximal trace of* $SD\}$*. A trace is maximal if it is not a proper prefix of any other trace.*

The same notions at the CPN level are recalled below.

Definition 7.4 (CPN Trace) *A trace of a CPN is a possibly infinite word* w*,* $w = m_1 \cdot m_2 \cdot m_3 \cdot \ldots$ *over the CPN alphabet* L_3 *iff there exists a sequence of places* $p_1 \cdot p_2 \cdot p_3 \cdot \ldots$ *over* P *of the same colour* $c \in \Sigma$*, a sequence of arcs* $a_1 \cdot a_1^{'} \cdot a_2 \cdot a_2^{'} \cdot \ldots$ *over* A*, a sequence of transitions* $\sigma = t_1 \cdot t_2 \cdot t_3 \cdot \ldots$ *over* T_n*, and a sequence of labelled transitions* $t_1^{'} \cdot t_2^{'} \cdot t_3^{'} \cdot \ldots$ *over* T_n^l *obtained from* σ *by removing the transitions without labels, such that* $m(p_1) \geq 1$*,* $node(a_i) = (p_i, t_i)$*,* $node(a_i^{'}) = (t_i, p_{i+1})$*, and* $l(t_i^{'}) = m_i$ *for all* $i \in \mathbb{N}$*.*

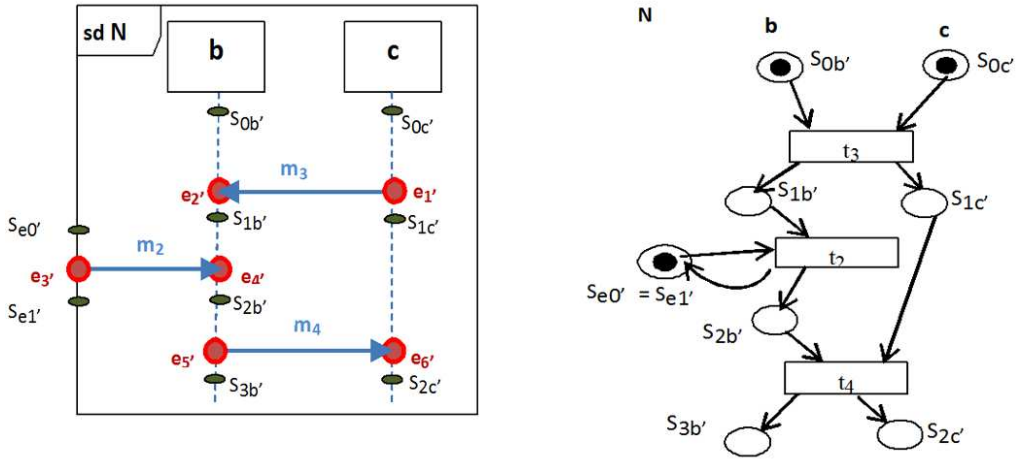
As indicated above, transitions without labels (effectively those used for denoting the beginning/end of fragments) are ignored in the trace.

Definition 7.5 (CPN Language) *The language of a CPN is the set* $\mathcal{L}_3(CPN)$ *of words over the alphabet* L_3*, where* $\mathcal{L}_3(CPN) = \{W|$ *W is a maximal trace of* $CPN\}$*, where a trace is maximal when it is not a proper prefix of any other trace.*

7.2.1 Language Equivalence of the Transformations

We defined the model transformations in Chapter 5 and 6 in such a way that SDs and CPNs use the same alphabet M. For example, consider the SD and the corresponding CPN shown in Figure 7.10. The alphabet used is $M = \{m_2, m_3, m_4\}$. Notice that for the CPN, we only depict the names of the net

transitions (here t_2, t_3, t_4 and not their labels $l(t_2) = m_2$, $l(t_3) = m_3$, and $l(t_4) = m_4$) and do not show the labels to keep the diagrams clearer. We obtain two traces for SD_N over the message labels such that $w_1 = m_3 \cdot m_2 \cdot m_4$ for instance $b \in I$ and $w_2 = m_3 \cdot m_4$ for $c \in I$. Similarly, for CPN_N we obtain (the same) traces over M given by words $w_1 = m_3 \cdot m_2 \cdot m_4$ for colour b and $w_2 = m_3 \cdot m_4$ for colour c.



(a) (b)

Figure 7.10: A simple SD_N and corresponding CPN_N.

As this simple example has shown, all traces for a given instance in a SD are preserved in the CPN for the matching colour by our SD-to-CPN transformation. More generally, we prove that the languages associated with a SD and corresponding CPN obtained by our transformation are *equivalent* also known as *strongly consistent* as follows. This result gives us a proof of the semantic correctness of our transformation.

Theorem 7.2 *Let SD be a sequence diagram and CPN be the corresponding coloured Petri net obtained following our transformation rules. If $\mathcal{L}_1(SD)$ is the set of words defined over the alphabet L_1 and $\mathcal{L}_2(CPN)$ the set of words defined over alphabet L_2. Then*

(1) $L_1 = L_2$ *and*

(2) $\mathcal{L}_1(SD) = \mathcal{L}_2(CPN)$.

Proof 7.1 *(1) The equality $L_1 = L_2$ is true by definition since both models use the same alphabet of message labels M. (2) To prove language equivalence $\mathcal{L}_1(SD) = \mathcal{L}_2(CPN)$ we show separately (i) $\mathcal{L}_1(SD) \subseteq \mathcal{L}_2(CPN)$ and (ii) $\mathcal{L}_2(CPN) \subseteq \mathcal{L}_1(SD)$.*

Case (i): We prove this directly, assuming that there is a word $w \in \mathcal{L}_1(SD)$ and showing that how necessarily $w \in \mathcal{L}_2(CPN)$. Let $w = m_1 \cdot m_2 \cdot m_3 \cdot \ldots$. Since $w \in \mathcal{L}_1(SD)$ there is a chain c for an instance $o \in I$ which determines w in SD. Let the chain be given by $c = s_1 \cdot e_1 \cdot s_2 \cdot e_2 \cdot \cdots \cdot s_k \cdot e_k \cdot s_{k+1} \ldots$. Through application of our transformation rules to c we obtain a sequence of places $s_1 \cdot s_2 \cdot \ldots s_k \cdot s_{k+1} \ldots$ over P of colour o; and for each event in the chain e_k if $\mu_o(m_k, e_k)$ is defined then there is a matching net transition $t_k \in T_n$; otherwise the event marks the beginning/end of a fragment and there is a $t_k \in T_n^{\neg l}$ (given by Rule 5.15 in Chapter 5).

Further we automatically obtain a sequence of transitions $\sigma = t_1 \cdot t_2 \cdot \cdots \cdot t_k \cdot t_{k+1} \ldots$ over T_n, and a sequence of labelled transitions $t_1^{'} \cdot t_2^{'} \cdot \cdots \cdot t_k^{'} \cdot t_{k+1}^{'} \ldots$ over T_n^l obtained from σ by removing the transitions without labels. By definition we also have an automatic sequence of arcs $a_1 \cdot a_1^{'} \cdot a_2 \cdot a_2^{'} \cdot \ldots a_k \cdot a_k^{'} \cdot a_{k+1} \cdot a_{k+1}^{'} \cdot \ldots$ over A. Since $m(s_1) \geq 1$, $node(a_i) = (s_i, t_i)$, $node(a_i^{'}) = (t_i, s_{i+1})$, and $l(t_i^{'}) = m_i$ for all $i \in \mathbb{N}$, we actually have a CPN trace obtained from c and by definition this trace $w \in \mathcal{L}_2(CPN)$.

Case (ii): We prove this by contradiction, that is, we assume that there is a word $w \in \mathcal{L}_2(CPN)$ such that $w \notin \mathcal{L}_1(SD)$ and show how this leads to a contradiction. Let $w = m_1 \cdot m_2 \cdot m_3 \cdot \ldots$. Since $w \in \mathcal{L}_2(CPN)$, by definition of a CPN trace there exists a sequence of places of the same colour $o \in \Sigma$, $p_1 \cdot p_2 \cdot p_3 \cdot \ldots$ over P, a sequence of arcs $a_1 \cdot a_1^{'} \cdot a_2 \cdot a_2^{'} \cdot \ldots$ over A, a sequence

of transitions $\sigma = t_1 \cdot t_2 \cdot t_3 \cdot \ldots$ *over* T_n*, and a sequence of labelled transitions* $t_1^{'} \cdot t_2^{'} \cdot t_3^{'} \cdot \ldots$ *over* T_n^l *obtained from* σ *by removing the transitions without labels, such that* $m(p_1) \geq 1$*,* $node(a_i) = (p_i, t_i)$*,* $node(a_i^{'}) = (t_i, p_{i+1})$*, and* $l(t_i^{'}) = m_i$ *for all* $i \in \mathbb{N}$*. Since* $w \notin \mathcal{L}_1(SD)$ *there is no chain (sequence of state locations and events) for instance* o *in* SD *which can lead to the sequence of places, transitions and arcs in the* CPN *following our transformation rules. We define this by induction on the length of the word* w*. Since initial state locations map onto places with initial marking, let us assume the problem lies at length* $k+1$*, that is, there is a chain* $c = p_1 \cdot e_1 \cdot p_2 \cdot e_2 \cdot \cdots \cdot p_k \cdot e_k \cdot p_{k+1} \cdot e_{k+1} \ldots$ *such that up to length* k *we would obtain a subword of* w *in the CPN, but the chain becomes invalid in step* $k+1$*. That means that* $p_{k+1} \notin next_o(e_k)$ *or* $e_{k+1} \notin next_o(p_{k+1})$*. The only transformation rules defining net transitions result from events involved in fragments or in local transitions, so if* c *is not a valid chain in SD the transformation from* $k+1$ *will also not be possible in the CPN which contradicts the assumption.* □

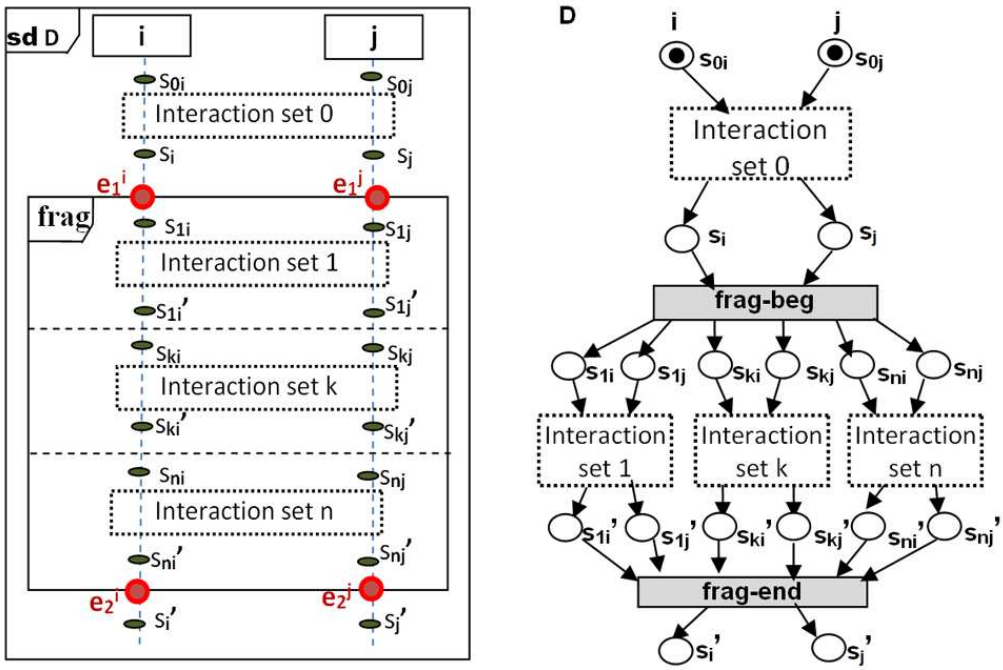
The above proof established the semantic correctness of the SD-to-CPN transformation by proving that the languages (sets of legal traces) associated with SDs and CPNs are equivalent under the transformation.

7.2.2 Correctness of Transformation Rules

This section proofs the correctness of the individual interaction fragment transformation rules.

Consider the correctness of Rule 5.15 that describes the transformation of a general interaction fragment.

Figure 7.11 shows the transformation of an arbitrary interaction fragment to a CPN, that synchronises the instances at the begining and end of the fragment behaviour.



(a) (b)

Figure 7.11: A sequence diagram with an interaction fragment (a) and the corresponding CPN (b).

Lemma 7.3 *Transformation rule of an arbitrary interaction fragment is behaviourally correct.*

Proof 7.2 *In the presence of an arbitrary interaction fragment behaviour, the alphabet of sequence diagrams (and CPNs) is as before over M. All we need to guarantee that the fragment transformation rule preserves the transformation of arbitrary words, i.e., if $w \in \mathcal{L}_1(SD)$ then $w \in \mathcal{L}_2(CPN)$.*

Consider, $x \in F$ be an interaction fragment in SD with $f(x) = (o, n)$, and $i \in j(x, k)$ be an arbitrary instance involved in the fragment for $1 \leq k \leq n$. Let $e_1, e_2 \in E_i$ denote the minimal and maximal event in $\overline{g(x)_i}$ respectively. Here, $s_k = min(\lambda_i(x, k))$ and $s_k^{'} = max(\lambda_i(x, k))$ with $e_1 \in next_i(s)$, $next_i(e_2) = \theta_i(x) = s^{'}$ and $next_i(e_1) = \{s_1, \ldots, s_n\}$, $e_2 \in next_i(s_k^{'})$. Let each operand k

contains a set of local transition $t_{1_k}, t_{2_k}, \ldots, t_{n_k}$.

Considering only the behaviour corresponding to the begining and the end of the fragment, the associated events do not correspond to a local transition and consequently do not contribute to the trace. The traces $w_k \in \mathcal{L}_1(SD)$ *for the behaviour within each operand can be derived based on different fragment types. Here, let us assume for an operand* k*, the default trace is* $w_k = m_{1_k} \cdot m_{2_k} \ldots m_{n_k}$*, where* $l(t_{j_k}) = m_{j_k}$*,* $1 \leq j \leq n$.

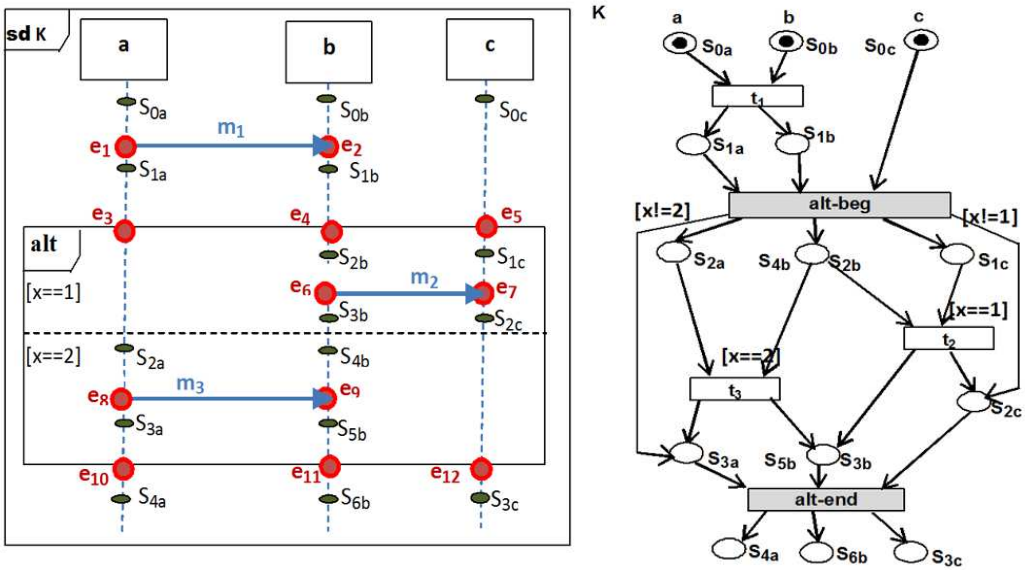
Applying Rule 5.15, the corresponding CPN *contains places* $s, s_1, \ldots, s_n, s_1^{'}, \ldots, s_n^{'}, s^{'} \in P$*, transitions* $t_{o\_beg}, t_{o\_end} \in T_n^{\neg l}$*, and arcs* $a_{i0}, a_{i1}, \ldots, a_{in}, a_{i1}^{'}, \ldots, a_{in}^{'}, a_{i0}^{'} \in A$ *such that* $node(a_{i0}) = (s, t_{o\_beg})$*,* $node(a_{ik}) = (t_{o\_beg}, s_k)$*,* $node(a_{ik}^{'}) = (s_k^{'}, t_{o\_end})$*, and* $node(a_{i0}^{'}) = (t_{o\_end}, s^{'})$.

Since t_{o_beg}, t_{o_end} *are unlabelled net transitions, they are not consider for the CPN word. The traces of the CPN* $w_k^{'} \in \mathcal{L}_2(CPN)$ *are based on the behaviour in the fragments. Here, the corresponding default trace can be derived as,* $w_k^{'} = m_{1_k} \cdot m_{2_k} \ldots m_{n_k}$*, where* $l(t_{j_k}) = m_{j_k}$*,* $1 \leq j \leq n$*. Here,* $\forall\, t_{j_k} \in T_n$ *are the labelled net transitions corresponding to the local transitions in each operand. These transitions are correctly transformed as the underlying languages are equivalent. Since* $w_k = w_k^{'}$*, Rule 5.15 preserves the same languages in both the CPN and the SD.*

Further, let us assume $w_k \neq w_k^{'}$ *and there exists a word* $w_k^{'} = m_\emptyset, m_{1_k} \cdot m_{2_k} \ldots m_{n_k}, m_\emptyset^{'}$*, assuming* $l(t_{o\_beg}) = m_\emptyset$ *and* $l(t_{o\_end}) = m_\emptyset^{'}$ *in the CPN. However,* $t_{o\_beg}, t_{o\_end} \in T_n^{\neg l}$ *and the labelling function* $l()$ *is not defined on them. This contradicts the assumption. Therefore, it is proven that Rule 5.15 is correct.* □

Consider the correctness of Rule 5.16 with alternative behaviour.

Consider SD_K and CPN_K shown in Figure 7.12 with alternative behaviour. Here, we can derive two traces over message labels such that $w_1 = m_1 \cdot m_2$ and



(a) (b)

Figure 7.12: A sequence diagram with alternative behaviour (a) and the corresponding CPN (b).

$w_2 = m_1 \cdot m_3$. Similarly, for CPN_K we can obtain equivalent traces $m_1 \cdot m_2$ and $m_1 \cdot m_3$ where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $m_3 = l(t_3)$. Here, the transition t_2 or t_3 executes based on the condition that evaluates to true.

Lemma 7.4 *Transformation rule of an alternative interaction fragment is behaviourally correct.*

Proof 7.3 *We need to guarantee that the alt rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* alt *behaviour.*

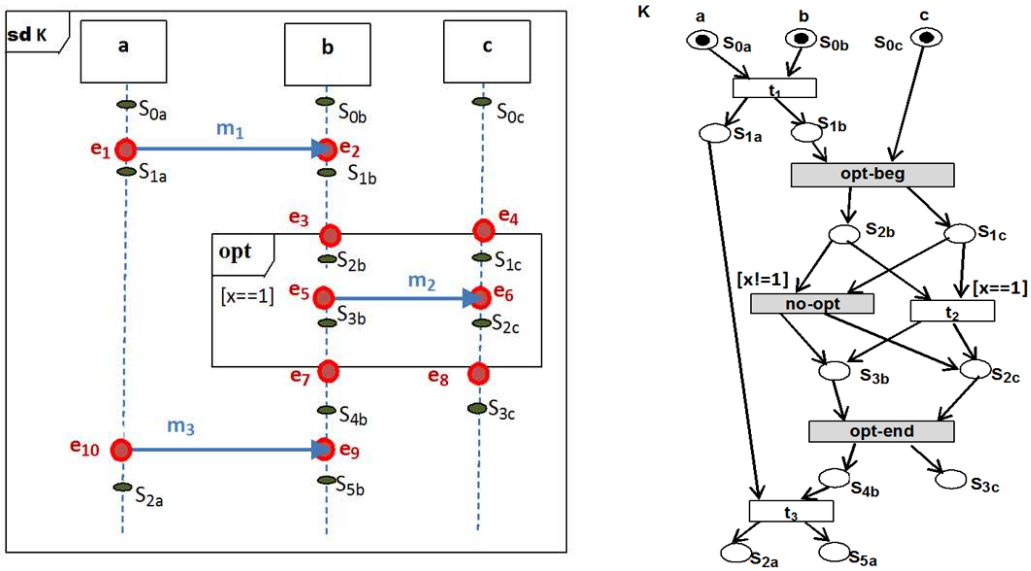
Consider, $x \in F$ *an interaction fragment in SD with* $f(x) = (alt, n)$*, and* $i \in j(x)$ *an arbitrary instance involved in the fragment and k is an operand such that* $1 \leq k \leq n$*. As explained in Proof 7.2, let the word* $w_k = m_{1_k} \cdot m_{2_k} \ldots m_{n_k}$*, where* $l(t_{j_k}) = m_{j_k}$*,* $1 \leq j \leq n$*, represents a trace within an operand in the alternative fragment and defined over the SD alphabet such that* $w_k \in \mathcal{L}_1(SD)$*.*

Similarly, the traces within the CPN can be derived as $w_k^{'} = m_{1_k} \cdot m_{2_k} \ldots m_{n_k}$, *where* $l(t_{j_k}) = m_{j_k}$, $1 \leq j \leq n$ *for* $w_k^{'} \in \mathcal{L}_2(CPN)$ *and* $w_k = w_k^{'}$.

Additionally, in an alt fragment the traces associated with each operand are mutually exclusive. i.e. $w_i \cap w_j = \emptyset$, *for* $1 \leq i, j \leq n$. *This behaviour is given by Rule 5.16 indicating transition executions are based on the associated condition that evaluates to true.*

Further, let us assume that w_i *and* w_j *are not mutually exclusive and there exist a word* $w \in w_i \cap w_j$ *in the CPN. Then it represents a default behaviour without any alternative behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent an alternative behaviour, which is a contradiction. Therefore, it is proven that Rule 5.16 is correct.* □

Consider the correctness of Rule 5.17 with optional behaviour.



(a) (b)

Figure 7.13: A sequence diagram with optional behaviour (a) and the corresponding CPN (b).

Consider SD_K and CPN_K shown in Figure 7.13 with optional behaviour.

Here, we can derive two traces over message labels such that $w_1 = m_1 \cdot m_2 \cdot m_3$ and $w_2 = m_1 \cdot m_3$. Similarly, for CPN_K we can obtain equivalent traces $m_1 \cdot m_2 \cdot m_3$ and $m_1 \cdot m_3$ where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $m_3 = l(t_3)$. Here, the transition t_2 executes based on the condition that evaluates to true.

Lemma 7.5 *Transformation rule of an option interaction fragment is behaviourally correct.*

Proof 7.4 *We need to guarantee that the opt rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* opt *behaviour.*

Consider, $x \in F$ *an interaction fragment in SD with* $f(x) = (opt, 1)$, *and* $i \in j(x)$ *an arbitrary instance involved in the fragment. As explained in Proof 7.2, let the words* $w_1 = m_1 \cdot m_2 \ldots m_n$, *where* $l(t_j) = m_j$, $1 \leq j \leq n$ *and* $w_2 = \emptyset$ *represent the trace within the optional behaviour and non-optional behaviour, respectively. These are defined over the SD alphabet such that* $w_1, w_2 \in \mathcal{L}_1(SD)$. *As given by Rule 5.17,* w_1 *is obtained when* $guard(x) = [C == True]$ *and* w_2 *otherwise.*

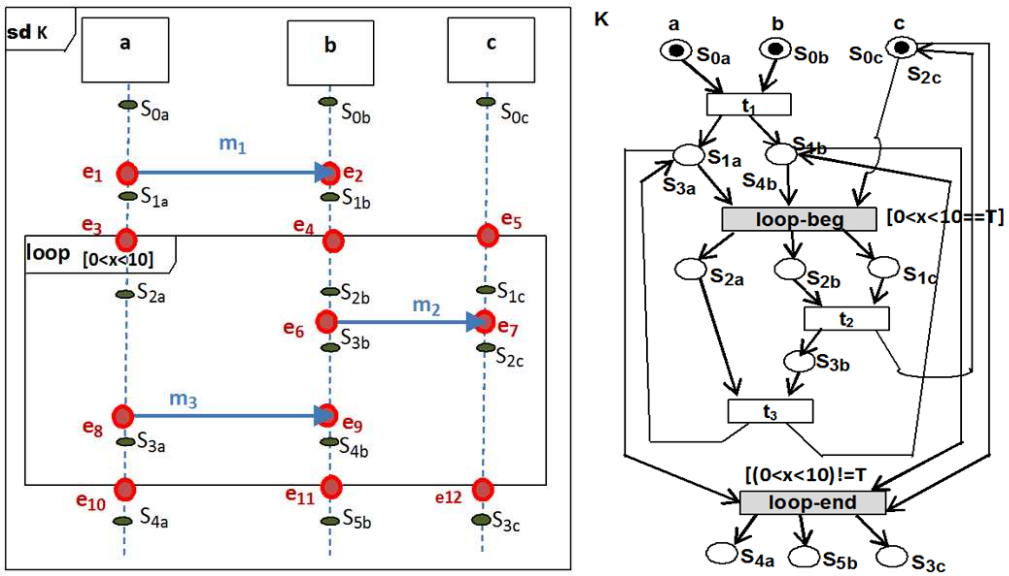
Similarly, considering the optional and non-optional behaviour only, the traces within the CPN $w_1^{'}, w_2^{'} \in \mathcal{L}_2(CPN)$, *can be derived as* $w_1^{'} = m_1 \cdot m_2 \ldots m_n$, *where* $l(t_j) = m_j$, $1 \leq j \leq n$ *and* $w_1^{'} = \emptyset$, *when* $guard(t_{no-opt}) = [C! = True]$. *Here,* $w_1^{'}, w_2^{'} \in \mathcal{L}_2(CPN)$ *and* $w_k = w_k^{'}$ *for* $k = 1, 2$.

Additionally, in an opt fragment the traces associated with optional and non-optional behaviour are mutually exclusive. i.e. $w_i \not\subseteq w_j$, *for* $i, j \in \{1, 2\}$. *This behaviour is given by Rule 5.17 indicating transition executions are based on the associated condition that evaluates to true.*

Further, let us assume w_i *and* w_j *are not mutually exclusive and there exist a word* $w_i \subseteq w_j$ *in the CPN. Then it represents a default behaviour without any optional behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$

the corresponding SD does not represent an optional behaviour, which is a contradiction. Therefore, it is proven that Rule 5.17 is correct. □

Consider the correctness of Rule 5.18 with iterative behaviour.



(a) (b)

Figure 7.14: A sequence diagram with iterative behaviour (a) and the corresponding CPN (b).

Consider SD_K and CPN_K shown in Figure 7.14 with iterative behaviour. Here, we can derive a trace over message labels such that $w_1 = m_1 \cdot \{m_2 \cdot m_3\}^*$. The associated chain is derived using the function *next* (Definition 3.8), where $next_b(S_{4b}) = e_4$ and this gives the repetitive behaviour. Similarly, for CPN_K we can obtain equivalent traces $m_1 \cdot \{m_2 \cdot m_3\}^*$ where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $m_3 = l(t_3)$. Here, the transition t_2 and t_3 execute repeatedly until the associated condition evaluates to true.

Lemma 7.6 *Transformation rule of an iterative interaction fragment is behaviourally correct.*

Proof 7.5 *We need to guarantee that the loop rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* loop *behaviour.*

Consider, $x \in F$ *an interaction fragment in SD with* $f(x) = (loop, 1)$*, and* $i \in j(x)$ *an arbitrary instance involved in the fragment. As explained in Proof 7.2, let the word* $w = \{m_1 \cdot m_2 \ldots m_n\}^*$*, where* $l(t_j) = m_j$*,* $1 \leq j \leq n$ *represents the trace within the iterative behaviour. This is defined over the SD alphabet such that* $w \in \mathcal{L}_1(SD)$*. Here,* $*$ *indicates the number of iterations that the interactions execute such that* $min \leq * \leq max$*, where* $c = [min \leq v \leq max]$ *for* $c \in guard(x)$ *and* $v \in X$*.*

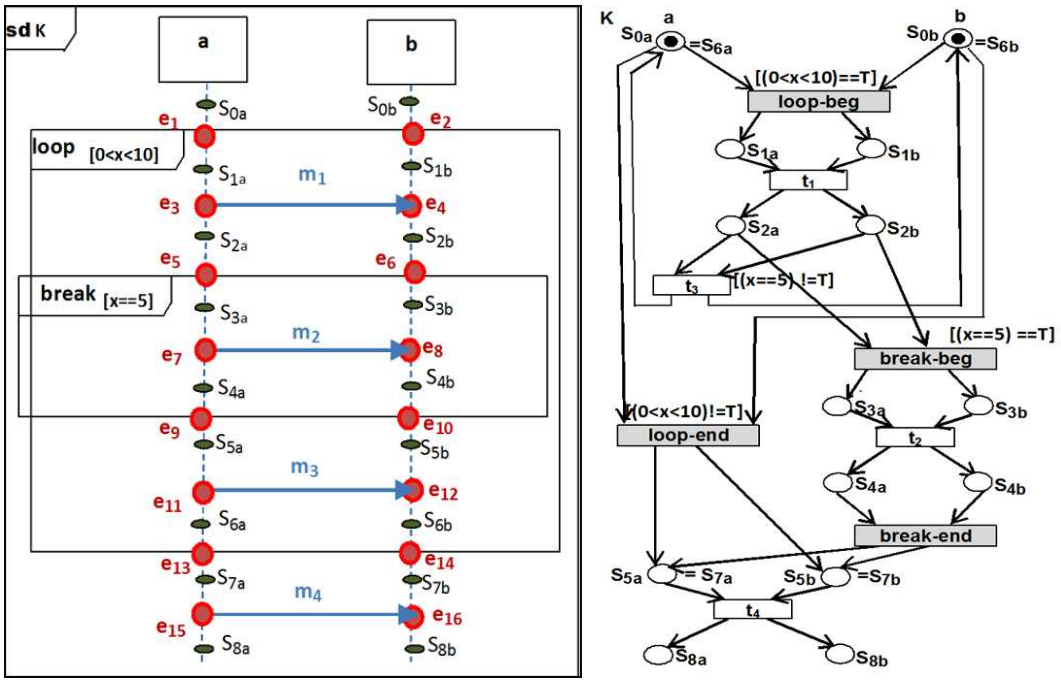
Similarly, considering the iterative behaviour only, the trace within the CPN $w^{'} \in \mathcal{L}_2(CPN)$*, can be derived as* $w^{'} = \{m_1 \cdot m_2 \ldots m_n\}^*$*, where* $l(t_j) = m_j$*,* $1 \leq j \leq n$ *for* $w^{'} \in \mathcal{L}_2(CPN)$ *and* $w = w^{'}$*. The behaviour indicates by this trace, executes repeatedly until the associated condition is false as given by Rule 5.18, ie.* $guard(t_{loop-end}) = [C! = True]$*.*

Further, let us assume w *is not an iterative trace and there exist a word* $w = m_1 \cdot m_2 \ldots m_n$ *in the CPN, i.e.* $v = 1$*. That indicates all transitions are executed with default behaviour and there is no iterative behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent a loop behaviour, which is a contradiction. Therefore, it is proven that Rule 5.18 is correct.* □

Consider the correctness of Rule 5.19 with break behaviour.

Here, two traces over message labels can be derived considering the break and non-break behaviour. The function *next* (Definition 3.8) is used to derive the underlying chain.

Consider SD_K and CPN_K shown in Figure 7.15 with *break* fragment that is nested in a *loop* fragment. The trace $w_1 = m_1 \cdot m_2 \cdot m_4$ applies when



(a) (b)

Figure 7.15: A sequence diagram with break behaviour: Case II (a) and the corresponding CPN (b).

the associated condition within the *break* fragment evaluates to true during the first iteration of the loop, and the trace $w_2 = \{m_1 \cdot m_3\}^* \cdot m_2 \cdot m_4$ applies whenever the condition evaluates to true during the iteration behaviour. Here, $next_b(e_{10}) = S_{7b}$. Similarly, for the break and non-break behaviour in CPN_K equivalent traces $m_1 \cdot m_2 \cdot m_4$ and $\{m_1 \cdot m_3\}^* \cdot m_2 \cdot m_4$ where $m_k = l(t_k)$ for $k = \{1, 2, 3, 4\}$ can be obtained. Here, the transition t_2, that includes in the *break* fragment fires, only when the associated condition evaluates to true: $guard(break - beg) = [C == True]$. Otherwise, the transitions within the break behaviour do not execute.

Lemma 7.7 *Transformation rule of a break interaction fragment is behaviourally correct.*

Proof 7.6 *We need to guarantee that the break rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* break *behaviour.*

Let $x, y \in F$ *be interaction fragments in SD with* $f(y) = (loop, 1)$, $f(x) = (break, 1)$, *such that* $h(y, 1) = x$ *and* $i \in j(y) \cap j(x)$ *an arbitrary instance involved in the fragment. For the break behaviour* $next(max(\overline{g(x)_i})) = \theta_i(y)$ *(Definition 3.8).*

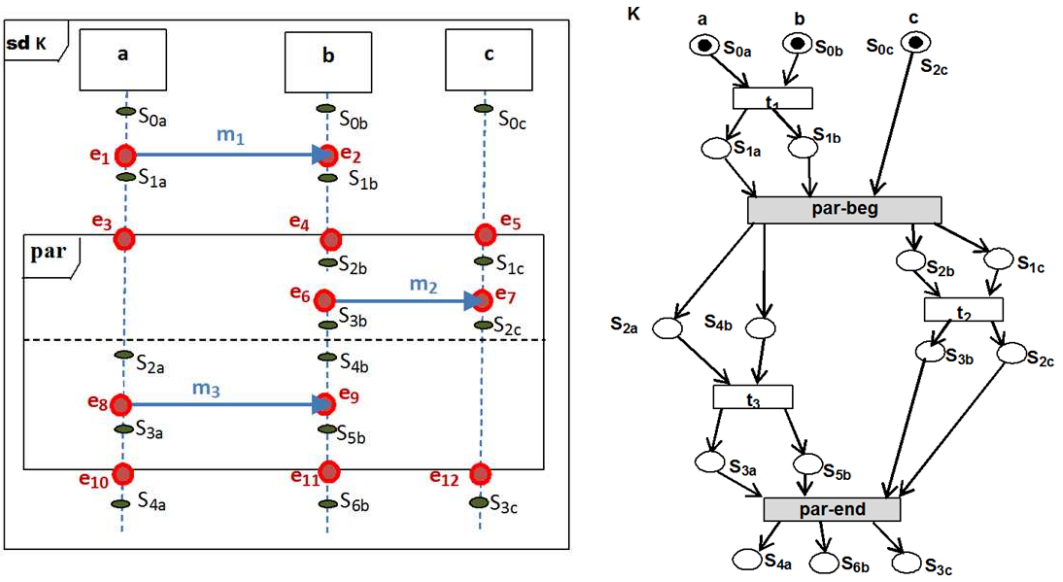
As explained in Proof 7.2, let $w_1 \in \mathcal{L}_1(SD) : w_1 = m_1 \cdot m_2 \ldots m_n \cdot m^{'}$ *represents a trace within the break behaviour when* $guard(x) = [C == True]$. *Here,* $l(t_j) = m_j$, $1 \leq j \leq n$ *and* $l(t^{'}) = m^{'}$ *for* $t^{'} = (e_1, m^{'}, e_2) : e_1, e_2 \in next(\theta_i(y))$. *Otherwise, the trace is same as the underlying loop behaviour and does not consider the interactions within the break fragment.*

As given by Rule 5.19, considering the break and non-break behaviour only, similar traces can be derived considering the corresponding elements in the CPN. Here, $w_1^{'} \in \mathcal{L}_2(CPN)$, *can be derived as* $w_1^{'} = m_1 \cdot m_2 \ldots m_n \cdot m^{'}$, *where* $l(t_j) = m_j$, $1 \leq j \leq n$ *when* $guard(t_{break-beg}) = [C = True]$. *Hence,* $w_1 = w_1^{'}$.

Further, assume there is a local transition after the break fragment and within the loop fragment : $t = (e, m, e^{'})$ *where* $e, e^{'} \in next(\theta_i(x))$. *let us assume there exist a trace* $w = m_1 \cdot m_2 \ldots m_n \cdot m$ *for* $w \in \mathcal{L}_2(CPN)$. *Then it represents a default behaviour without any break behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent a break behaviour, which is a contradiction. Therefore, it is proven that Rule 5.19 is correct.* □

Consider the correctness of transformation with parallel behaviour.

The operator *par* has a natural representation with the CPN model, which supports parallelism using Rule 5.15. Consider Figure 7.16 with parallel behaviour. The traces in SD_K can be derived as $w_1 = m_1 \cdot m_2 \cdot m_3$ and $w_2 = m_1 \cdot m_3 \cdot m_2$. Similarly, for CPN_K we can obtain equivalent traces



(a) (b)

Figure 7.16: A sequence diagram with parallel behaviour (a) and the corresponding CPN (b).

where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $m_3 = l(t_3)$. i.e. transitions t_2 and t_3 can be interleaved in any way.

Lemma 7.8 *Transformation rule of a parallel interaction fragment is behaviourally correct.*

Proof 7.7 *We need to guarantee that the transformation with parallel behaviour preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$.

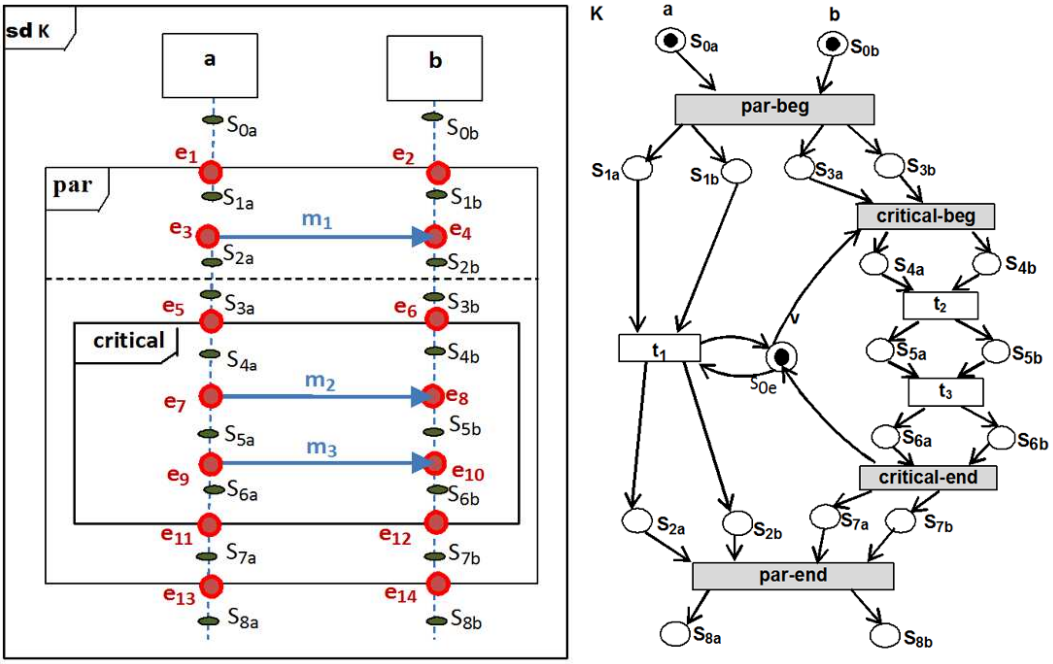
Consider, $x \in F$ *an interaction fragment in* SD *with* $f(x) = (par, n)$, *and* $i \in j(x)$ *an arbitrary instance involved in the fragment and* k *is an operand such that* $1 \leq k < n$. *Let* $q_k \in \mathbb{N}$ *be the number of transitions in a given operand* k. *A set of chains can be derived from this behaviour using the function* $next()$ *in such a way that, event occurrences of different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.*

As explained in Proof 7.2, and considering parallel behaviour only, let there an arbitrary word $w = m_{k_1} \dots m_{k_{(j)}} \dots m_{(k+1)_r} \dots m_{k_{(j+1)}} \dots m_{(k+1)_{(r+1)}} \dots : \forall k$ *and* $j, r < q_k$*. This represents a trace within the par fragment and defined over the SD alphabet such that* $w \in \mathcal{L}_1(SD)$*.*

Similarly, the traces within the CPN can be derived as $w' = m_{k_1} \dots m_{k_{(j)}} \dots m_{(k+1)_r} \dots m_{k_{(j+1)}} \dots m_{(k+1)_{(r+1)}} \dots : \forall k,\ j, r < q_k,$ *where* $l(t_{a_b}) = m_{a_b},\ \forall a \in k,\ \forall b \in j, r$*. Here,* $w' \in \mathcal{L}_2(CPN)$ *and* $w = w'$*.*

Hence, the operator par has a natural representation with the CPN model and proven that the transformation is correct. □

Consider the correctness of Rule 5.20 with critical behaviour.



(a) (b)

Figure 7.17: A sequence diagram with critical behaviour (a) and the corresponding CPN (b).

Consider SD_K and CPN_K shown in Figure 7.17 with *critical* fragment that

is nested in a *par* fragment. Here, a set of traces over message labels can be derived considering the critical behaviour, in such a way that the interactions within the critical behaviour cannot be interleaved with other parallel interactions in any way. Here, we can derive two traces over message labels such that $w_1 = m_1 \cdot m_2 \cdot m_3$ and $w_2 = m_2 \cdot m_3 \cdot m_1$, where m_2 is always followed by m_3. Equivalent traces can be derived in CPN, where $m_k = l(t_k)$ for $k = \{1, 2, 3\}$.

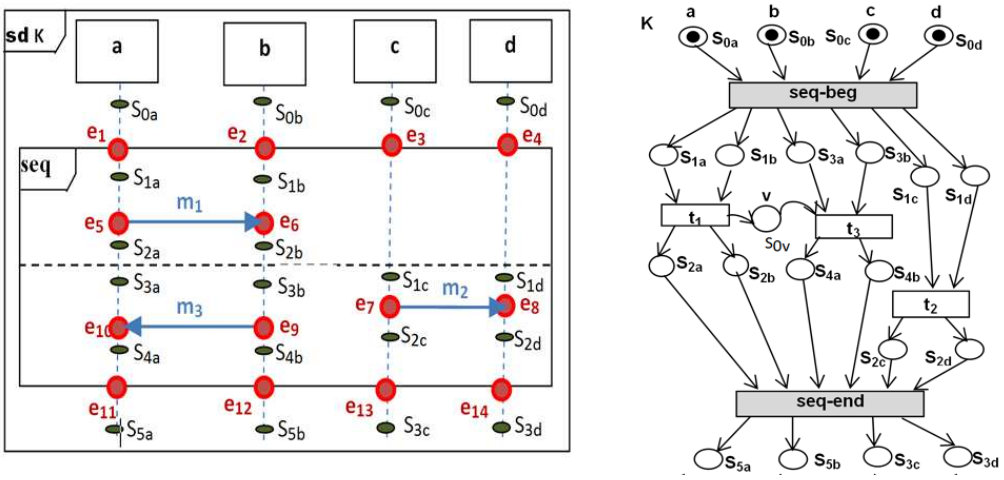
Lemma 7.9 *Transformation rule of a critical interaction fragment is behaviourally correct.*

Proof 7.8 *We need to guarantee that the critical rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* critical *behaviour.*

Let $x, y \in F$ *be interaction fragments in* SD *with* $f(y) = (par, n)$, $f(x) = (critical, 1)$, *such that* $h(y, 1) = x$ *and* $i \in j(y) \cap j(x)$ *an arbitrary instance involved in the fragment. As explained in Proof 7.2, let* $w_1 \in \mathcal{L}_1(SD)$ *:* $w_1 = m_1 \cdot m_2 \ldots m_n$ *represents a trace within the critical behaviour. As given by Rule 5.19, considering the critical behaviour only, a similar trace can be derived considering the corresponding elements in the CPN. Here,* $w_1^{'} \in \mathcal{L}_2(CPN)$, *can be derived as* $w_1^{'} = m_1 \cdot m_2 \ldots m_n$, *where* $l(t_j) = m_j$, $1 \leq j \leq n$.

Further, assume there exist a trace $w = m_1 \cdot m_2 \ldots m \ldots m_n \cdot m$ *for* $w \in \mathcal{L}_2(CPN)$, *where another transition :* $l(t) = m$ *is interleaved within the trace of a critical behaviour. Here, the interactions within a critical region are not treated as atomic and has been interrupted by another interaction. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent a critical behaviour, which is a contradiction. Therefore, it is proven that Rule 5.20 is correct.* □

Consider the correctness of Rule 5.21 with sequence behaviour.



(a) (b)

Figure 7.18: Sequence diagram with sequential behaviour and corresponding CPN.

Consider SD_K and CPN_K shown in Figure 7.18 with sequence behaviour. We can derive two traces over message labels such that $w_1 = m_1 \cdot m_2 \cdot m_3$ and $w_2 = m_2 \cdot m_1 \cdot m_3$. These traces define local causality inside and between operands of the fragment, when they share same instances. Similarly, for CPN_K we can obtain equivalent traces where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $m_3 = l(t_3)$.

Lemma 7.10 *Transformation rule of a sequence interaction fragment is behaviourally correct.*

Proof 7.9 *We need to guarantee that the seq rule preserves the transformation of arbitrary words, i.e., if $w \in \mathcal{L}_1(SD)$ then $w \in \mathcal{L}_2(CPN)$ by applying the transformation rule with* seq *behaviour.*

Consider, $x \in F$ an interaction fragment in SD with $f(x) = (seq, n)$, and $i \in j(x,k) \cap j(x,(k+1))$ for $k \in \mathbb{N}$ and $k < n$, an arbitrary instance involved in the operand k and $(k+1)$. Let $q_k \in \mathbb{N}$ be the number of transitions in a

given operand k that shares the same instances. A set of chains can be derived from this behaviour in such a way that, event occurrences of different operands can be interleaved only if the involved instances are mutually exclusive.

As explained in Proof 7.2, let there be an arbitrary word $w = m_{k_1} \dots m_{k_{(j)}} \dots m_{(k+1)_r} \dots m_{k_{(j+1)}} \dots m_{(k+1)_j} \dots m_{(k+1)_{(j+1)}} \dots$, $\forall k,\ j, r < q_k$. *Here the associated transitions are involved in instances: for* $t_{kj} = (e_1, m_{kj}, e_2)$, $t_{kr} = (e_3, m_{kr}, e_4)$ *in such a way that* $e_1, e_2 \in E^a \cup E^b$ *and* $e_3, e_4 \in E^c \cup E^d$. *This represents a trace within the seq fragment and defined over the SD alphabet such that* $w \in \mathcal{L}_1(SD)$.

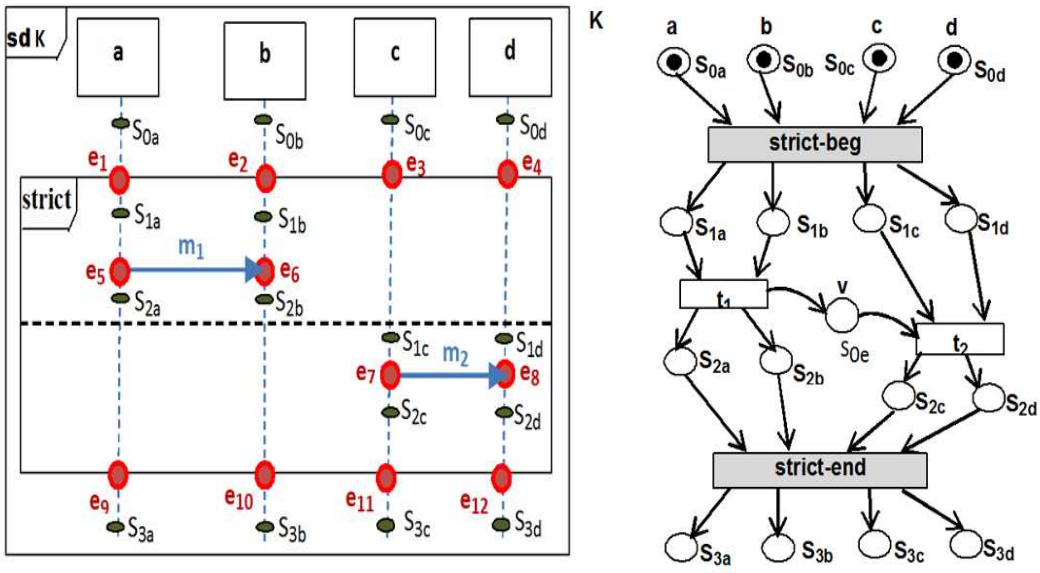
Similarly, the traces within the CPN can be derived as $w^{'} = m_{k_1} \dots m_{k_{(j)}} \dots m_{(k+1)_r} \dots m_{k_{(j+1)}} \dots m_{(k+1)_j} \dots m_{(k+1)_{(j+1)}} \dots$, $\forall k,\ j, r < q_k$, *where* $l(t_{k_j}) = m_{k_j}$ *and* $l(t_{k_r}) = m_{k_r}$ *for* $w_k^{'} \in \mathcal{L}_2(CPN)$ *and* $w_k = w_k^{'}$.

Further, let us assume a word in CPN such that $w_1 = m_{k_1} \cdots m_{k_{(j)}} \dots m_{(k+1)_j} \dots m_{k_{(j+1)}} \dots m_{(k+1)_{(j+1)}} \dots$, *such that the labels that correspond to the transitions between the operands, are interleaved even when they share the same instances. Then it represents a default behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent a sequence behaviour, which is a contradiction. Therefore, it is proven that Rule 5.21 is correct.* □

Consider the correctness of Rule 5.22 with strict behaviour.

Consider SD_K and CPN_K shown in Figure 7.18 with sequence behaviour. We can derive only one trace over message labels such that $w = m_1 \cdot m_2$. This trace defines a strict execution order between the interactions between the operands, even the involved instances are mutually exclusive. Similarly, for CPN_K we can obtain equivalent trace where $m_1 = l(t_1)$ and $m_2 = l(t_2)$.

Lemma 7.11 *Transformation rule of a strict interaction fragment is behaviourally correct.*



(a) (b)

Figure 7.19: A sequence diagram with strict behaviour (a) and the corresponding CPN (b).

Proof 7.10 *We need to guarantee that the strict rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* strict *behaviour.*

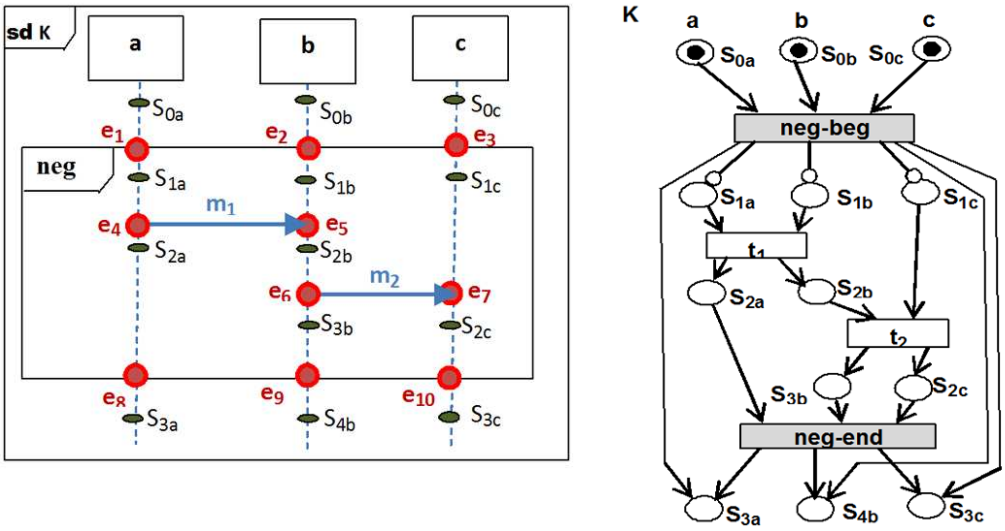
Consider, $x \in F$ *an interaction fragment in SD with* $f(x) = (strict, n)$*, and* k *is an operand such that* $1 \leq k \leq n$*. Let* $a, b \notin j(x,k) \cap j(x,(k+1))$ *for* $k < n$*, be mutually exclusive arbitrary instances involved in the operand* k *and* $(k+1)$*. Let* $q_k \in \mathbb{N}$ *be the number of transitions in a given operand* k*. A chain can be derived from this behaviour in such a way that, event occurrences in a operand execute before the event occurences in the next operand, and so on, and imposes a strict execution order between the behaviour of operands.*

As explained in Proof 7.2, let there be an arbitrary word $w = m_{k_1} \ldots m_{k_{(j)}} \ldots m_{k_{(j+1)}} \ldots m_{(k+1)_j} \ldots m_{(k+1)_{(j+1)}} \ldots$*,* $\forall k,\ j < q_k$*. This represents a trace within the strict fragment and defined over the SD alphabet such that* $w \in \mathcal{L}_1(SD)$*.*

Similarly, the traces within the CPN can be derived as $w^{'} = m_{k_1} \ldots m_{k_{(j)}} \ldots m_{k_{(j+1)}} \ldots m_{(k+1)_j} \ldots m_{(k+1)_{(j+1)}} \ldots$, $\forall k,\ j < q_k$, *where* $l(t_{k_j}) = m_{k_j}$ *for* $w_k^{'} \in \mathcal{L}_2(CPN)$ *and* $w_k = w_k^{'}$.

Further, let us assume a word in CPN such that $w_1 = m_{k_1} \cdots m_{k_{(j)}} \ldots m_{(k+1)_j} \ldots m_{k_{(j+1)}} \ldots m_{(k+1)_{(j+1)}} \ldots$, *such that the labels that correspond to the transitions between the operands, are interleaved even when they do not share the same instances. Here the corresponding transitions are involved in instances: for* $t_{kj} = (e_1, m_{kj}, e_2)$, $t_{(k+1)r} = (e_3, m_{(k+1)r}, e_4)$ *in such a way that* $e_1 \cup e_2 \in E^a$ *and* $e_3 \cup e_4 \in E^b$ *for* $j, r < q_k$. *Then it represents a default behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent a strict behaviour, which is a contradiction. Therefore, it is proven that Rule 5.22 is correct.* □

Consider the correctness of Rule 5.25 with negative behaviour.



(a) (b)

Figure 7.20: A sequence diagram with negative behaviour (a) and the corresponding CPN (b).

Consider SD_K and CPN_K shown in Figure 7.20 with negative behaviour.

Since it specifies a behaviour that must not occur, we can derive a trace over message labels such that $w_1 = \emptyset$ for the *neg* fragement. Similarly, for CPN_K we can obtain an equivalent trace $\emptyset$.

Lemma 7.12 *Transformation rule of a negative interaction fragment is behaviourally correct.*

Proof 7.11 *We need to guarantee that the neg rule preserves the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}_1(SD)$ *then* $w \in \mathcal{L}_2(CPN)$ *by applying the transformation rule with* neg *behaviour.*

Consider, $x \in F$ *an interaction fragment in SD with* $f(x) = (neg, 1)$. *As explained in Proof 7.2, let the word* $w = \emptyset$ *represents the trace within the negative behaviour. This is defined over the SD alphabet such that* $w \in \mathcal{L}_1(SD)$.

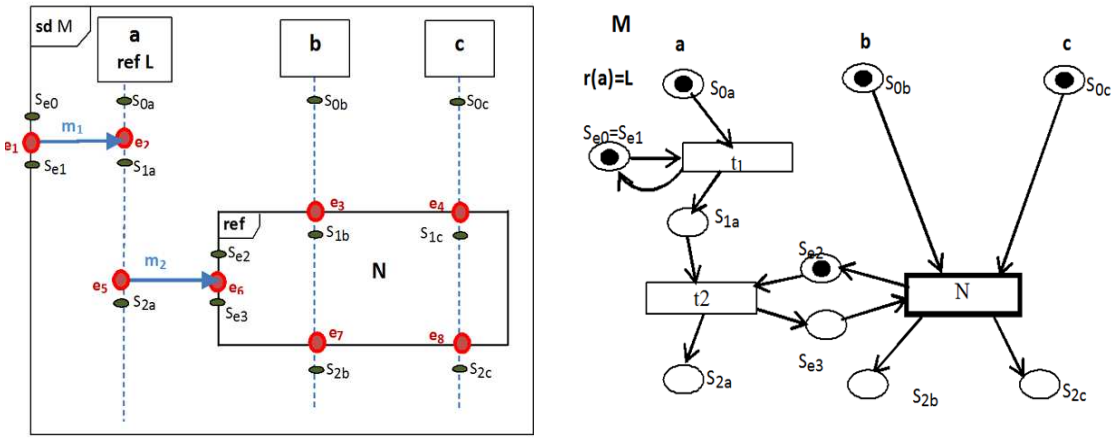
Similarly, considering the negative behaviour only, the trace within the CPN $w^{'} \in \mathcal{L}_2(CPN)$, *can be derived as* $w^{'} = \emptyset$ *for* $w^{'} \in \mathcal{L}_2(CPN)$ *and* $w = w^{'}$.

Further, let us assume there exist a word $w = m_1 \cdot m_2 \ldots m_n$ *in the CPN. That indicates all transitions are executed with default behaviour and there is no negative behaviour. i.e. for the obtained CPN :* $CPN = \tau(SD)$ *the corresponding SD does not represent a negative behaviour, which is a contradiction. Therefore, it is proven that Rule 5.25 is correct.* □

The remaining transformations for ignore (Rule 5.23), and assert (Rule 5.24) behaviours do not affect the traces of the corresponding CPN as they affect only on the status of the CPN places.

7.2.3 Language Equivalence of the Hierarchical Transformations

We analyse the implications of the decomposition rules on the result of language equivalancy proved in Section 7.2.1 as follows.



(a) (b)

Figure 7.21: A Sequence diagram with reference behaviour and corresponding CPN.

Theorem 7.13 *Adding reference mechanisms to the model transformation, preserves the equivalence established in Theorem 7.2.*

Proof 7.12 *In the presence of decomposition mechanisms, the alphabet of sequence diagrams (and CPNs) changes and is defined over* $M \cup \mathcal{N}$. *All we need to guarantee that the ref rules preserve the transformation of arbitrary words, i.e., if* $w \in \mathcal{L}(SD)$ *then* $w \in \mathcal{L}(CPN)$ *by applying the transformation rules with* interaction use *behaviour.*

Indeed, a word $w = m_1 \cdot m_2 \cdot N \cdot m_4 \ldots$ *over the SD alphabet belongs to* $w \in \mathcal{L}(CPN)$ *if we have a sequence of transitions* $\sigma = t_1 \cdot t_2 \cdot t_3 \cdot t_4 \ldots$ *where* $l(t_i) = m_i$ *for* $i = 1, 2, 4$ *and* $l(t_3) = N$. *If this were not the case, then the problem lies in* t_3 *because the remainder transitions are correctly transformed according to our previous result. However,* t_3 *cannot be a normal labelled net transition as there is no underlying local transition in the SD which maps onto it, and hence* $l(t_3) = K$. *Because SDs and CPNs match diagram names in the interaction-use rule we consequently have to have* $K = N$, *and hence*

$w \in \mathcal{L}(CPN)$. □

Consider SD_M and CPN_M shown in Figure 7.21 with reference behaviour. Here, we can derive a trace over message labels and diagram names such that $m_1 \cdot m_2 \cdot N$ for instance $a \in I$. Similarly, for CPN_M we can obtain an equivalent trace $m_1 \cdot m_2 \cdot N$ for colour a where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $N = l(t_N)$.

7.2.4 Language Equivalence of the Parametric Transformations

Finally, we can establish the semantic correctness of the parametric M2M transformations by proving that the languages (sets of legal traces) associated with SDs and CPNs variants are equivalent under the transformation. Here, we explore what it means to consider timing or stochastic annotations over a SD and resulting CPN variants. For our parametric transformations, semantic correctness is given as follows.

Theorem 7.14 *For sets of timing and stochastic annotations $\mathcal{T}$ and $\mathcal{S}$ defined over SD, and arbitrary subsets $\Gamma \subseteq \mathcal{T}$ and $\Psi \subseteq \mathcal{S}$. The following strong consistency or language equality result holds over parametric transformations: $par(\Gamma)(\mathcal{L}(SD)) = \mathcal{L}(CPN_\Gamma)$ and $par(\Psi)(\mathcal{L}(SD)) = \mathcal{L}(CPN_\Psi)$.*

Proof 7.13 *Adding annotations of different kinds to a SD means that each word in the language $w = m_1 \cdot m_2 \cdot m_3 \ldots$ is injected with some additional parameter at different places determined by the annotations. In other words, applying the parametric transformation on a SD changes its underlying alphabet accordingly. For instance, $L_\Psi = M \cup \Psi$ in the stochastic case. The legal traces in both SD and CPN variant languages, however, are still in essence the legal traces from the basic transformation (without parameters: $par()$) and consequently the strong consistency result obtained earlier is preserved with the timing and stochastic rules guaranteeing the correct mapping of annotations between $w \in par(\Gamma)(\mathcal{L}(SD))$ and $w \in \mathcal{L}(CPN_\Gamma)$ - similarly for $par(\Psi)$.* □

For SD_K in Figure 7.22, a word over $L_{\mathcal{S}}$ is given by $w = m_1 \cdot m_2 \cdot (m_3, (r_1, r_2)) \ldots$ for instance $b \in I$ where $r_1 = 5$ and $r_2 = 8$. Similarly, considering the labelled net transitions in CPN_K we obtain the equivalent trace $w = m_1 \cdot m_2 \cdot m_3(5, 8) \ldots$ for colour b where $m_1 = l(t_1)$, $m_2 = l(t_2)$ and $m_3 = l(t_3)$.

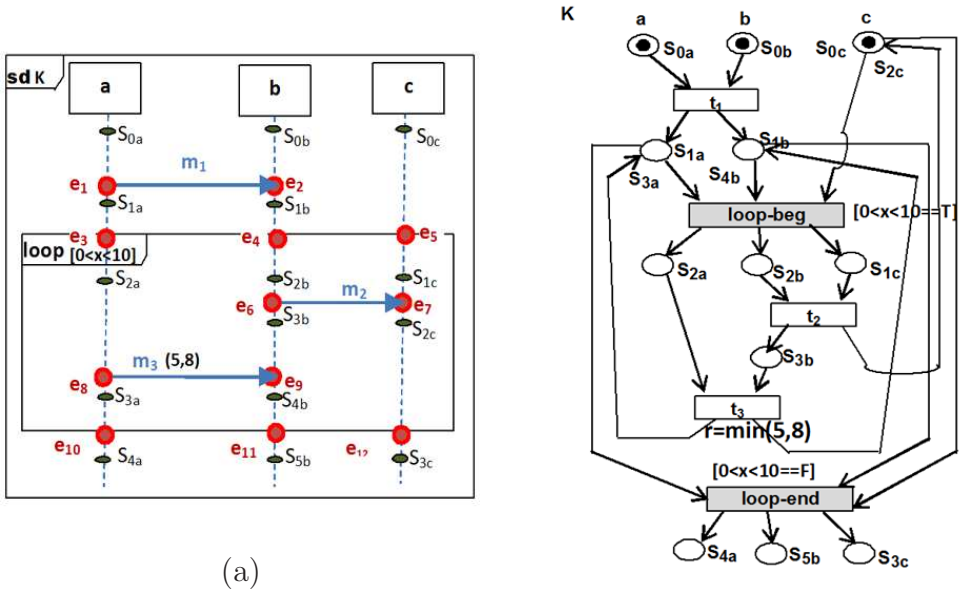


Figure 7.22: Sequence diagram with stochastic data and corresponding SCPN obtained by $par(\mathcal{S})$

7.2.5 Bisimulation Preservation

We have seen that our model transformation rules reflect a one-to-one correspondence between the elements in the source and target models. Moreover, and since the theorems above prove that the language associated with a source model is equivalent to the language of the target model obtained by transformation, we can also express semantical correctness of model transformations using *bisimulation*. Bisimulation is an equivalence relation between two mod-

els that defines whether two models have the same behaviour. In our case, we can define a natural notion of SD bisimulation based on traces as follows.

Definition 7.6 (SD Bisimulation) *Let SD_1 and SD_2 be two different sequence diagrams with the same set of object instances I and alphabet L. SD_1 and SD_2 are bisimilar, written $SD_1 \sim SD_2$, iff $\mathcal{L}_1(SD_1) = \mathcal{L}_1(SD_2)$.*

Overall, two sequence diagrams are bisimilar if their observable behaviour is the same regardless of how they were modelled (using interaction fragments, and so on). The same notion can be stated for CPNs leading to CPN bisimulation. Interestingly, our transformation preserves the notion of bisimulation from the source model to the target model.

Theorem 7.15 (Bisimulation Preservation) *Let SD_1 and SD_2 be bisimilar sequence diagrams, i.e., $SD_1 \sim SD_2$. The corresponding CPNs obtained by transformation, CPN_1 and CPN_2, are bisimilar $CPN_1 \sim CPN_2$.*

The proof follows directly from Theorem 7.2 and additional theorems for language equivalence from above.

7.3 Concluding Remarks

This Chapter has described the significance of having correct model transformations between models in MDD. The main contribution consists of establishing that our SD-to-CPN model transformation is strongly consistent focusing on semantic correctness and completeness. We also give a brief description of the approach generally adopted for showing (syntactical) correctness of model transformations using graph-based mechanisms. Usually model transformation rules are defined using Triple Graph Grammars (TGGs) with the added benefit that graph theoretic results can be explored to prove certain properties

of the transformation. However, the nature of these transformation rules only enables syntactical correctness, which is not sufficient for behavioural models. Our previously formally defined transformation rules are an alternative description to rules given with TGGs, and in this chapter we only show a few examples of how our rules can be given in a TGG style.

For behavioural model transformations a proof of semantic correctness is essential. We go one step further, by proving not only that the behaviour of the source model is preserved in the target model, but also that there are no additional behaviours possible in the target model by transformation. In other words, we establish a one-to-one correspondence between the legal traces of source and target models. The proof for semantical correctness is given in steps adding new constructs such as reference and parametric transformation incrementally. The parametric transformation includes extensions on both source and target models to address real-time and stochastic behaviour. We reflect on how a notion of bisimulation for both source and target models is preserved by transformation with our approach.

8 Chapter 8 : Support for Automated Model Transformation

The model transformation defined in this thesis is presented as a set of rules with a formal syntax and a denotational semantics, which in particular facilitates its proof of correctness. However, such a representation is not directly usable by developers, expect an automated model transformation to be directly embedded into the Integrated Development Environments (IDEs) and/or Computer-Aided Software Engineering (CASE) tools they are accustomed with. In order to facilitate this we have investigated and partially developed a prototype tool (SD2CPN tool) for the integrated and automated model transformation from a SD to a CPN.

In another point of view, a case study is an ideal methodology to examine a system. In the context of this thesis, case study-based examples enable a rigorous understanding of the model transformations and aim to generalise across a larger domain of application.

This chapter starts by explaining the architecture of the SD2CPN tool, which transforms a SD into an equivalent CPN that can be analysed through existing CPN tools. Section 8.2 describes the meta-models used for the front-end and back-end of the prototype tool. This includes the classes and their relationships that are used to implement the GUI (Graphical User Interface) of the tool as well as the actual transformation rules from a SD to a CPN. Section 8.3 explains the GUI of this prototype tool considering the input and output of the tool in graphical format.

Also we define text-based grammar for SDs and CPNs. Section 8.4 addressed these grammars based on Backus-Naur Form (BNF) [Reniers, 1998]. The textual input and output of the tool can be used to integrate the trans-

formations with the existing SD and CPN modelling tools. Moreover, Section 8.5 explains the implementation of the transformation rules. Only the basic transformations were incorporated in this prototype tool with the main aim of introducing an integrated tool with an IDE for these transformations. However, the tool architecture supports convenient extensions to include complex transformations and can hence be extended to incorporate all the defined rules of this thesis.

Further, Section 8.6 validates the applicability of the model transformation defined in this thesis using examples as case-studies.

8.1 SD2CPN Tool Design

We explore the possibility of developing a prototype tool that supports the present model-driven transformation framework. The basic theoretical aspects of the defined transformation rules were implemented in the SD2CPN tool. This tool inputs a SD and outputs the corresponding CPN, and the both models can represent graphically and/or in an equivalent textual notation. The tool can be applied generally to any software system and the textual notations can be used to integrate our tool more directly with other existing modelling tools. The main aim of this tool is to show how the transformation rules can be implemented.

Figure 8.1 shows an overview of the interactions between a user and the SD2CPN tool. A user can model a SD and the tool converts it to the corresponding CPN using the transformation rules defined. User interaction with the SD2CPN tool is based on the direct manipulation of either a graphical or a textual representation of a model. The GUI of the tool supports techniques such as tool palettes and marking menus. Thus the user is aided with drag and drop capabilities to model a SD. Alternatively, a user can also represent

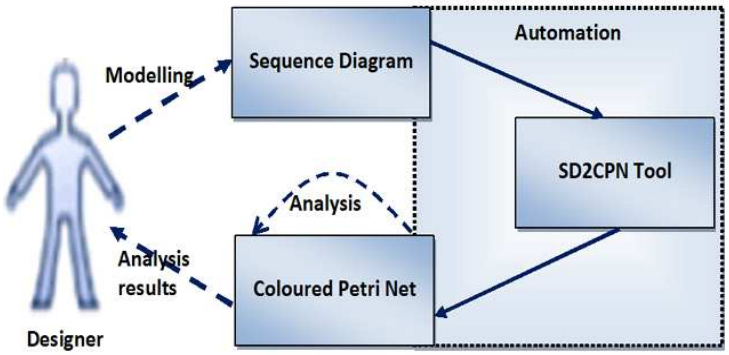


Figure 8.1: An overview of the designer interaction with the tool.

a SD textually using a textual notation (the grammar of the textual notation is defined in Section 8.4 in BNF format) with the same expressiveness as the more commonly used graphical notation.

The SD2CPN tool generates a formal representation of a SD according to the user input and generates the corresponding CPN in both graphical and textual formats. The synthesised CPN can be used to check system properties manually or automatically using existing tools and the results of the analysis returned to the user. When analysing a CPN model, the execution and object flow of the system can be illustrated by simulating the tokens as they are passed from a transition to another. Thus, CPN simulation can be shown to the user in order to reproduce expected scenarios of system behaviour and hence be used to validate the UML SD model.

At this point we should also add that at present the results returned to the designer are directly related to the CPN and assume an understanding of the CPN model and notation by the designer. Ideally this process would be transparent and the results of the analysis should be given in the context of the original model. Work on such a tool is however beyond the scope of the present thesis.

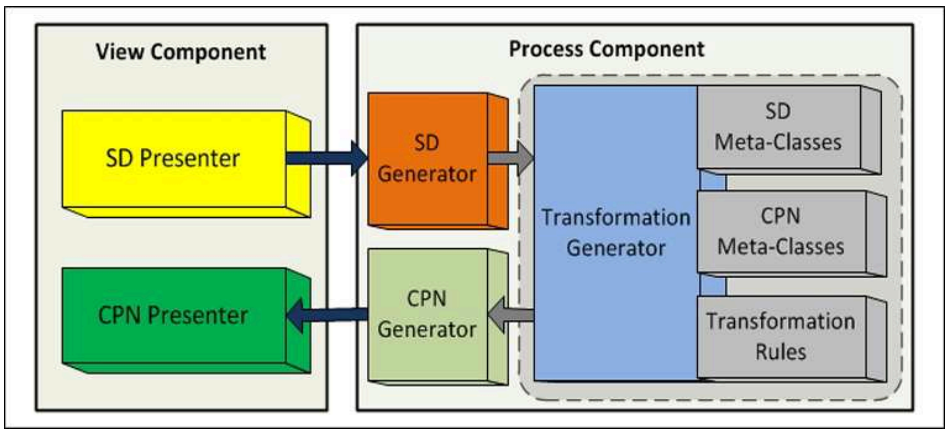


Figure 8.2: The high level architecture of the SDCPN tool.

Figure 8.2 shows the architecture of the SD2CPN tool. Using a component-based modular architecture, the tool is designed with two major components: *View Component* and *Process Component*. *View Component* constitutes the front-end of the tool that is visible to the user, and *Process Component* is constitutes the back-end of the tool with core transformation implementations.

Given a SD as an input, *SD Generator* generates an equivalent textual representation for the SD model and passes that representation to *Transformation Generator*. The underlying theory behind *Transformation Generator* is based on the model transformation framework SiTra (Simple Transformer) [Ameedeen and Bordbar, 2008, Akehurst et al., 2006].

The meta-models of a SD and a CPN (defined in Section 8.2), and the formal transformation rules (given in Chapter 5) have been considered for the implementation of *Process Ccomponent* of the SDCPN tool. A separate class accessible through a common interface, is implemented for each element of the meta-models (*SD Meta-Classes*, *CPN Meta-Classes*) and for each transformation rule. The *Transformation Rules* component contains the Java functions that map a given meta-class of a SD to the corresponding meta-class of the CPN. The component *Transformation Generator* performs the necessary

mapping operations with the use of the meta-classes and the formal rules.

This architectural framework provides easy route for the implementation with the capability of accessing the separate classes through a common interface. i.e., the Java class implementations of the model elements and the rules are independent from the implementation of the transformation process. Additionally, *Transformation Generator* keeps track of the partial ordering among the elements of the models. After that, *CPN Generator* builds the corresponding CPN using the outcome of the transformations and passes that data to *CPN presenter* in order to represent the resulting CPN graphically or textually as required by the user.

Figure 8.3 depicts an outline of the model transformation process within the SD2CPN tool that complies with MDD. The component *Transformation Generator* uses the transformation rules to implement how various elements of the SD meta-model are mapped to the elements of the CPN meta-model. This is carried out automatically via *Transformation Generator* and the entire process is commonly referred to as the Model Transformation Framework (MTF). A typical MTF requires three inputs: a source meta-model, a destination meta-model and a set of transformation rules. For each instance of the source meta-model, *Transformation Generator* executes the rules to create an instance of the destination meta-model.

The graphical model representations (SD and CPN) shown in the diagram are the screenshots of the SD2CPN tool which conform to their meta-models. Further, the text-based BNF model representations can be used for the input and the output instead of the graphical representations, and can facilitate possible integrations with existing modelling tools.

This prototype tool is implemented in *Java* on the *NetBeans* environment, version 6.7 of the *Windows* platform. All GUI based classes in *View Component*

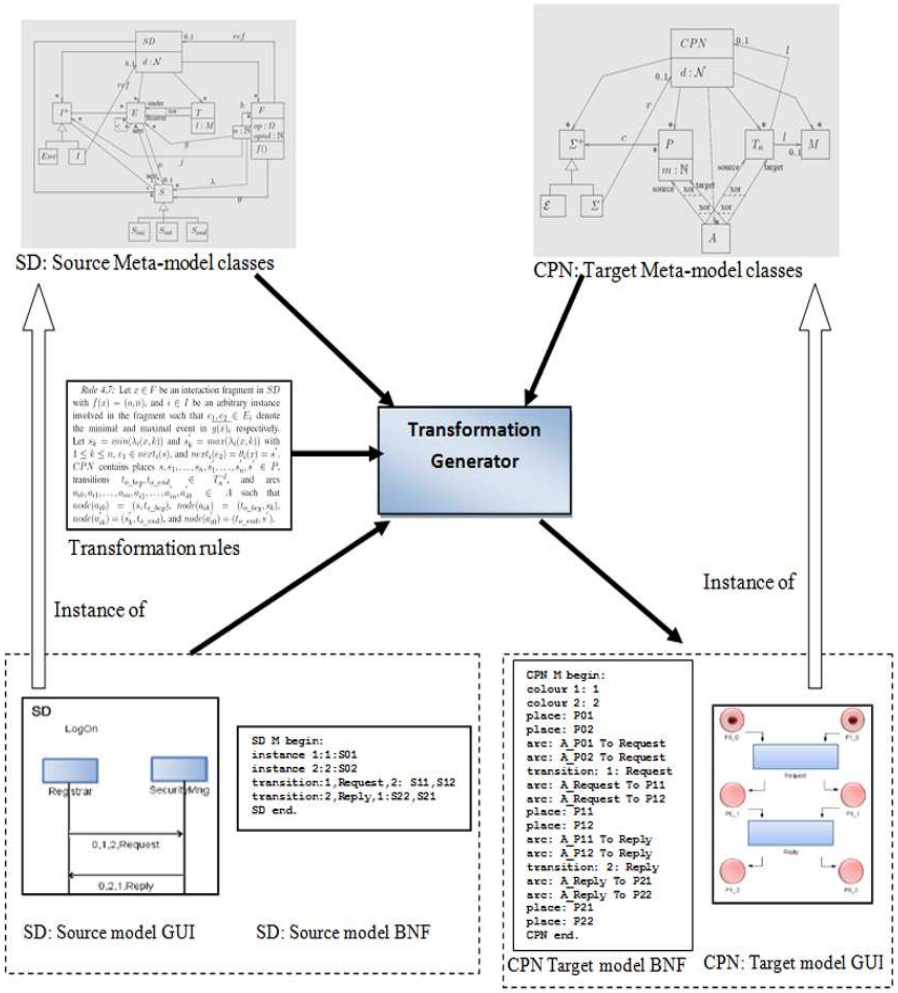


Figure 8.3: The SD2CPN tool framework complies with MDD.

(front-end) are implemented with the *look and feel* feature in *NetBeans Visual Library API* and *Utilities API*. The prototype tool is being developed to satisfy the requirements of optimal model transformations, while aiming for an enhanced graphical interface. All the other classes in *Process Component* are implemented in *Java* general programming language.

8.2 SD2CPN Meta-models

Performing a model transformation by taking one or more models as the input and producing one or more models as the output requires a clear understanding of the abstract syntax and the semantics of the source and the target models. Meta-modelling is a key concept in MDA that defines the abstract syntax of the models and the inter-relationships between the model elements [Kleppe et al., 2003, Naumenko and Wegmann, 2002]. Tool implementation based on the meta-models of the graphical modelling languages is beneficial in several ways [Ouardani et al., 2006, dos S. Soares and Vrancken, 2008, Laleau and Polack, 2008]. Thus, a precise meta-model is a prerequisite for performing automated model transformations [T.Mens and Grop, 2006].

The meta-models defined in this section comply with the Meta-Object Facility (MOF) language in MDA [OMG, 2003,OMG, 2011a]. These meta-models are represented using class diagrams, where each class in the meta-model describes a set of objects that share the same specifications of features, constraints, and semantics. A class is a classifier whose features are attributes and operations, where attributes indicate the properties owned by the class and an operation get invoked on an object and may cause changes to the values of the attributes. The relationships between the classes are shown using different association types and a class may play a role in an association. Further, associations contain multiplicity elements that specify the allowable cardinalities for an instantiation of an element and embed the lower and the upper bounds of objects. The following sections describe the meta-models used for the back-end and the front-end of the SD2CPN tool.

8.2.1 Back-end Meta-models

This section describes the meta-models used for the back-end of the SD2CPN prototype tool, including the meta-models for a SD, a CPN and for the transformation process.

UML sequence diagram (SD) is itself a designed and architected system [OMG, 2011a]. Figure 8.4 shows a meta-model of the model elements and their relationships in a SD that conform with formal Definition 3.1 and Definition 3.2 defined in Chapter 3. The input model for the SD2CPN tool is an instance of this SD meta-model.

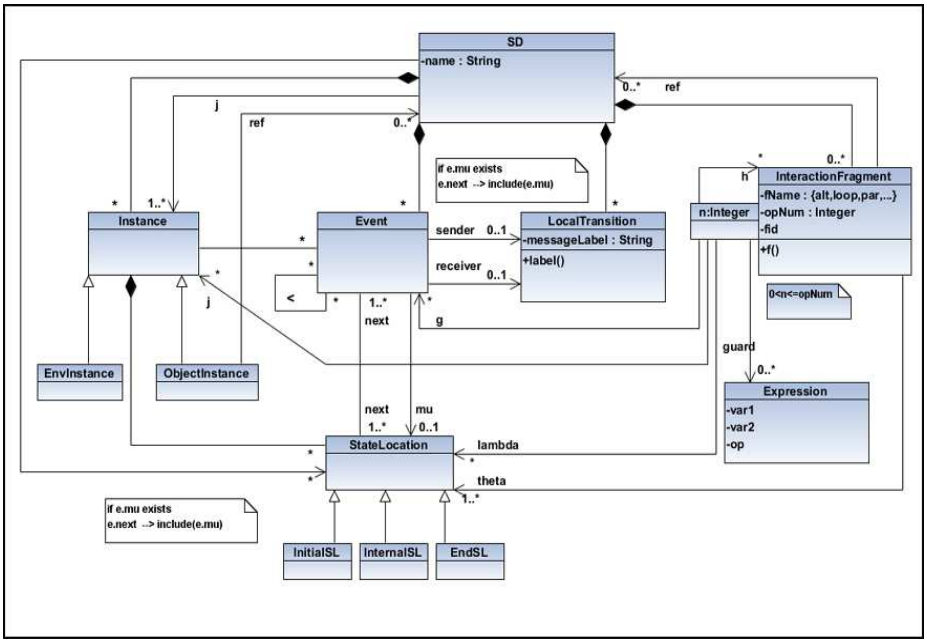


Figure 8.4: The SD meta-model of the SD2CPN tool.

The SD meta-model shown in Figure 8.4 comprises with the constructs that show both basic and complex behaviours. The main elements of a SD such as instances, events, local transitions and interaction fragments are associated with the class SD using composite relationships. The name of the model is included as an attribute in the class SD. Class $Instance$ has associations to the

classes *Event* and *StateLocation* to represent the events and state locations belonging to an instance respectively. There are two specialisations for an instance, namely object ($ObjectInstance$) and environment ($EnvInstance$) instances.

Additionally, the class *StateLocation* is specialised into three classes that represent initial, internal and end state locations. The occurrences of the class *Event* are partially ordered and the association *next* is defined for each class *Event* and *StateLocation*. Also, an *Event* instance may or may not be associated to a *StateLocation* instance through role *mu* (matching formally defined function μ). Further, the class $LocalTransition$ contains an attribute *messageLabel* and an operation *label*() that (re)assigns a message label to a local transition. An instance of class $LocalTransition$ has a sender and a receiver event which is given by the two associations (and corresponding rolenames) from $LocalTransition$ to *Event*.

An interaction fragment in a SD shows a complex behaviour and consists of one or more operands. The class $InteractionFragment$ contains the name ($fName$), number of operands ($opNum$) and the identifier (fid) of the fragment as its attributes. An operand number in the class $InteractionFragment$ (indicates by the qualifier n) has associations with the classes $Instance$, *Event*, *StateLocation* and nested $InteractionFragments$ as indicated by the associations j, g, *lambda* and h, respectively. These functions have been formally defined in Chapter 3.

Also, an operand may associates with *Expression* that contains variables and an operator. Further, the interaction reference behaviour that refers to another SD is given by the association ref and link with the classes, named $InteractionFragment$ and SD.

Figure 8.5 shows the CPN meta-model used for the SD2CPN prototype

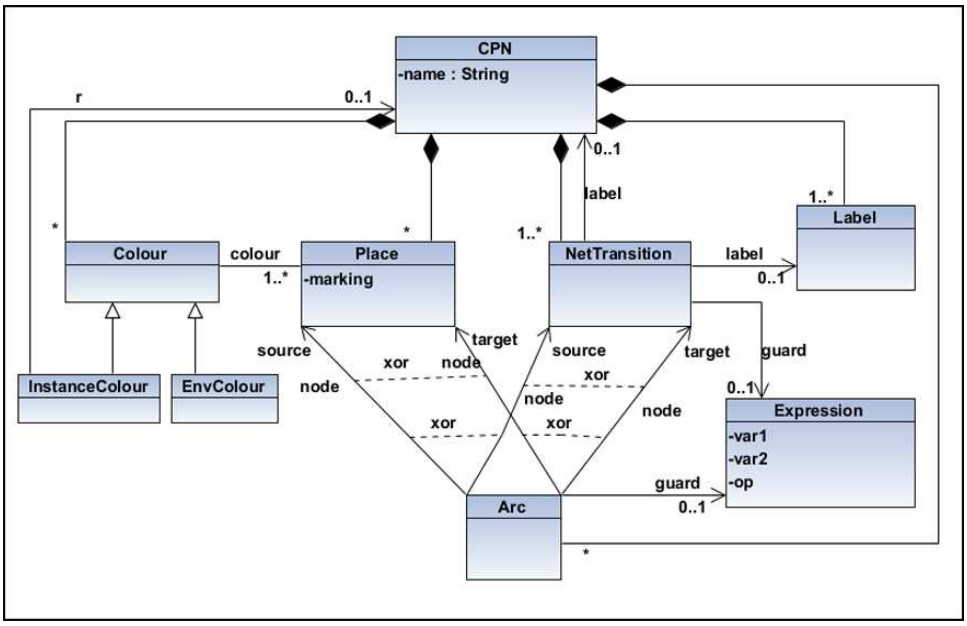


Figure 8.5: The CPN meta-model of the SD2CPN tool.

tool and it conforms the CPN Definition 4.1 in Chapter 3. The name of the CPN contains as an attribute of the class CPN. The main classes $Place$, $NetTransition$, Arc, $Label$ and $Colour$ are associated with CPN using composite relationships. Similarly to the class $Instance$ in the SD meta-model, there are two specialisations of the class $Colour$, namely $InstanceColour$ and $EnvColour$.

The class $Place$ links with the class $Colour$ using the association *colour* that denotes its colour type. Also it contains attributes *marking* that represents the number of tokens associated with the place, and *status* that shows the status of the place (can be complete, safe, etc.). The class $NetTransition$ connects with the class $Label$ with the association *label* and may link with the class $Expression$ using the association *guard*. Further, the class Arc associates with one instance of the classes $Place$ and $NetTransition$ using the *node* relationship, and there is an `xor` (exclusive or) constraint to reflect the

formally defined *node* function. For example, an arc cannot have a place as both source and target. The class *Expression* contains the variables and operators as its attributes. The allowable cardinalities for an instantiation of an element are shown using the multiplicities on each association.

In this implementation, the SD and the CPN meta-model classes are contained in separate packages (*SDMetaModel* and *CPNMetaModel* in Figure 8.6) and later access by the component *TransformGenerator* in the SD2CPN tool. Both of these SD and CPN meta-model do not contain the variants of the models, including timed and stochastic aspects. However, these meta-models can be extended with additional behaviours.

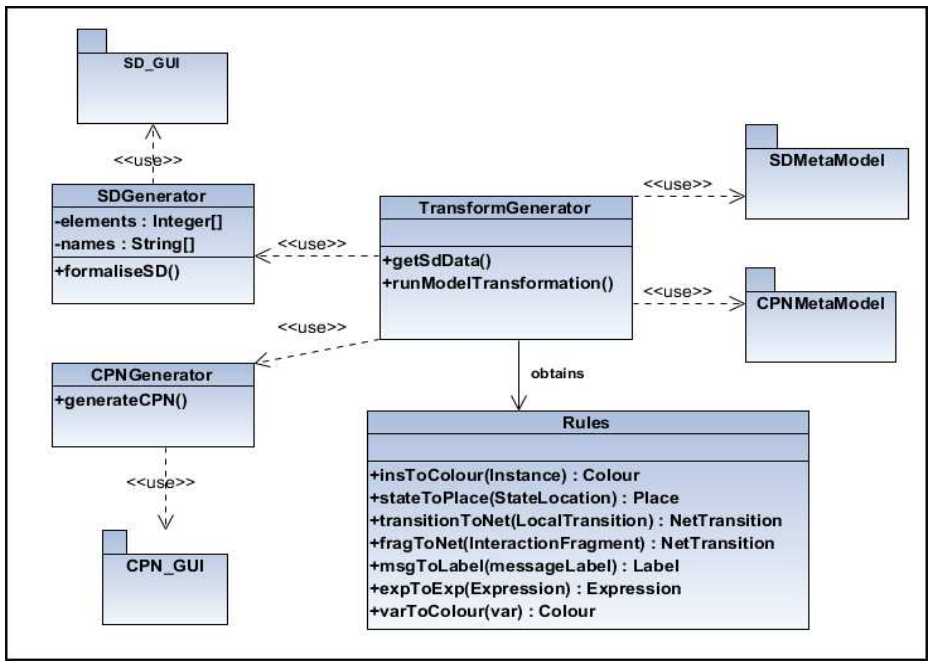


Figure 8.6: The meta-model for the Transform Generator of the SD2CPN tool.

Figure 8.6 shows the meta-model used for the transformation process itself, which is implemented with Java applications. The class *SDGenerator* retrieves the text-based data of the input model from the *SD_GUI* package that corresponds to the front-end SD GUI represented in Figure 8.7. The

$SDMetaModel$ and $CPNMetaModel$ packages correspond to the SD and CPN meta-models described in Figure 8.4 and Figure 8.5, respectively. Each transformation rule defined in Chapter 5 is implemented as an operation in the class *Rule* and made available to the class $TransformGenerator$.

The class $TransformGenerator$ uses the formalised input model from the class $SDGenerator$, meta-model elements from the packages $SDMetaModel$ and $CPNMetaModel$, and obtains the transformation rules implemented in the class *Rules* in order to perform the $runModelTransformation()$ operation. Here, class $TransformGenerator$ uses elements of the SD meta-model and the CPN meta-model to assign the relevant input and output model data, respectively. The operation $runModelTransformation()$ maps a given object in the SD model to the corresponding object in the CPN model by calling the relevant transformation rule. This process is applied for each element of the input SD model and at the completion, the class $CPNGenerator$ generates the target CPN in a text-based format. Then it uses the CPN_GUI package (in Figure 8.8) to display the CPN model in a visual representation.

8.2.2 Front-end Meta-models

This section explains the meta-model used for the front-end of the SD2CPN tool including the GUI representation for the SD and CPN models. The meta-model for the GUI of the SD and the CPN are included in the package SD_GUI and CPN_GUI, respectively, and imported by the $Transform\ Generator$ meta-model shown in Figure 8.6.

Figure 8.7 shows the meta-model of the GUI that facilitates to draw a SD. This meta-model is based on the Net Beans Modules, Visual Library API and Utilities API that supports for palette components with drag and drop capabilities and scene implementations with action handlers. Here, the

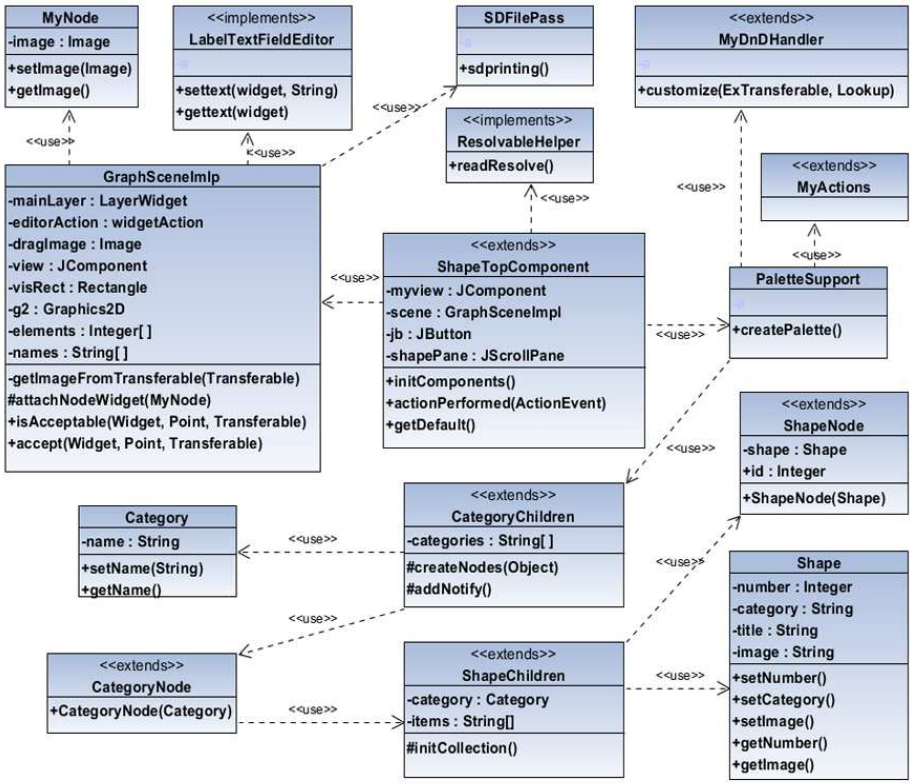


Figure 8.7: The meta-model for the Front-end GUI of the SD2CPN tool.

libraries provide a set of reusable widgets that facilitates visualisation and the pluggable components that are declared as interfaces or abstract classes.

The class *ShapeTopComponent* is an extension of a *TopComponet*, which is a Java Open IDE library that facilitates to display the components in the GUI. The *ShapeTopComponent* constructor initialise the components of the GUI such as *JScrollPane* and its layout. The class *ShapeTopComponent* creates an instance of the class *GraphSceneImpl* that holds visual data. Then it assigns the created scene to the view, *myView*, which is a *JComponent*, and sets its view to *viewpoint*.

Here, *JComponent* is a *javax swing* library that is used to display com-

ponents in the GUI. The class *ShapeTopComponent* also uses *PaletteSupport* that facilitates the drag and drop SD components to the drawing area. Further, *ShapeTopComponent* uses a class *ResolvableHelper* that implements *Serializable*, which is a Java input output interface. This facilitates application state serialisation at node level. Although, this does not directly relevant to the SD2CPN tool, it automatically selects the last selected node, in case of application restart, and included as a user supportive action.

The class *PaletteSupport* uses the class *CategoryChildren* to create nodes of the palette, sets the palette root and handles drag and drop facilities with *lookup* components using the class *MyDnDHandler*. All the other related classes are designed to increase the level of abstraction of the design in such a way that the class *Category* contains the name of the category, *CategoryChildren* contains an array of categories, *CategoryNode* uses *ShapeChildren* that contains an array of items, which includes the path for each image component of the palette, *Shape* contains the shape properties, and *ShapeNode* is a node with a shape. For this prototype tool, we have included the SD images for only testing purpose. The associated class *MyAction* is an extension of *PaletteAction* Java class that sets the palette actions to null.

The class *GraphSceneImpl* is an extension of a class *GraphScene*, which is a Java Visual API. This class facilitates functions such as add node widgets to the layers and perform move and zoom actions on widgets. (widget refers to constructs of java user interfaces). The method *getImageFromTransferable*() is a helper method that supports to retrieve the image from the transferable. The method *attachNodeWidget*() defines a new widget and is called automatically by the method *accept*() when an element from the palette is dropped to the scene. This method sets the image that is retrieved from the node, contains actions to set and change the label of the widget and implements actions

to move the widget in the scene.

Finally, the new widget is added to *LayerWidget*, which is a transparent pane that facilitates drag and drop functionality and returns the widget to the scene. Here, *Transferable* is a Java AWT interface for classes that can be used to provide data for a transfer operations. The methods *isAcceptable*() and *accept*() are called by the constructor. When a palette element is dragged over the scene, the method *isAcceptable*() determines whether the element is acceptable to the scene. If the method is acceptable, then the method *accept*() is called to get the image from the transferable.

Further this class uses the class *LabelTextFieldEditor* that implements *TextFieldInplaceEditor*, which is an interface for text-field based in-place editor in the NetBeans visual API to edit the labels of the images. The class *MyNode* contain an image as a node and uses by the class *GraphSceneImpl* to make instances of a given image. Finally, *GraphSceneImpl* facilitates to retrieve data from the GUI components and uses the class *SDFilePass* to write the data to a text file that can be used for further processing based on the SD model.

The meta-model of the graphical representation for the generated CPN is shown in Figure 8.8. This meta-model is based on a Java Swing application and facilitates to display the CPN GUI with look and feel features.

The class *CPNDraw* facilitates to represent the CPN graphically, based on the generated CPN by *Process Component* of the SD2CPN tool. The class *CPNDraw* extends a *JPanel*, which is a java general purpose lightweight container that is used to hold the widgets of Java Swing. This helps to position and structure the components based on the code. This class contains a *JFrame* as a top-level container and uses the class *GraphSceneImpl* that draws the CPN representation.

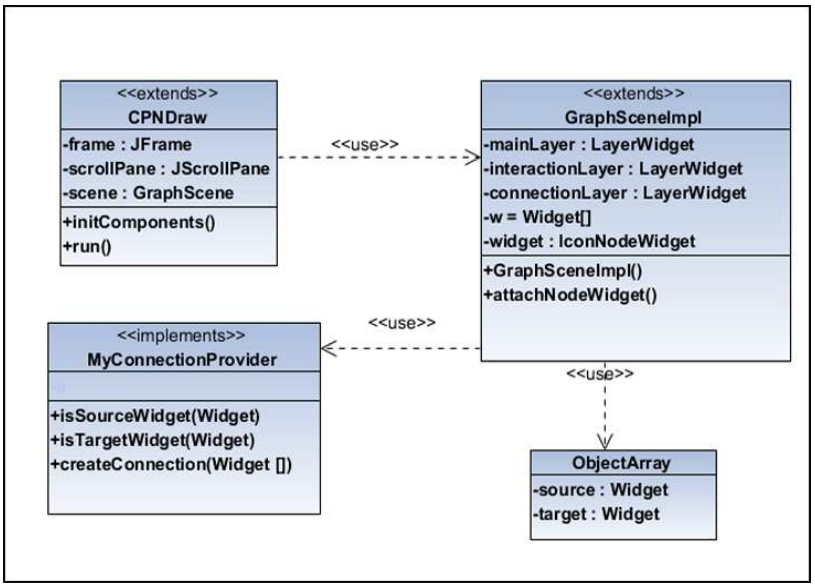


Figure 8.8: The meta-model for the Front-end CPN GUI of the SD2CPN tool.

The class *GraphSceneImpl* extends the class *GraphScene* (a NetBeans visual API) that holds and manages graph-oriented models with nodes and edges. It contains different *LayerWidgets* (NetBeans visual API class for a transparent widget), namely main layer, interaction layer and connection layer. Each layer carries out a different function: main layer for node widgets, connection layer for edge widgets and interaction layer for temporary widgets created/used by actions. The class constructor adds the nodes and sets the preferred locations for the nodes that correspond to the elements of the CPN.

The method *attachNodeWidget* is responsible for creating the widget, setting an image for it, adding it into the scene and returning it from the method. Further, the class *GraphSceneImpl* uses the source and the target widgets in the class *ObjectArray* and uses the class *MyConnectionProvider* to show the links between the places and the net transitions of the CPN.

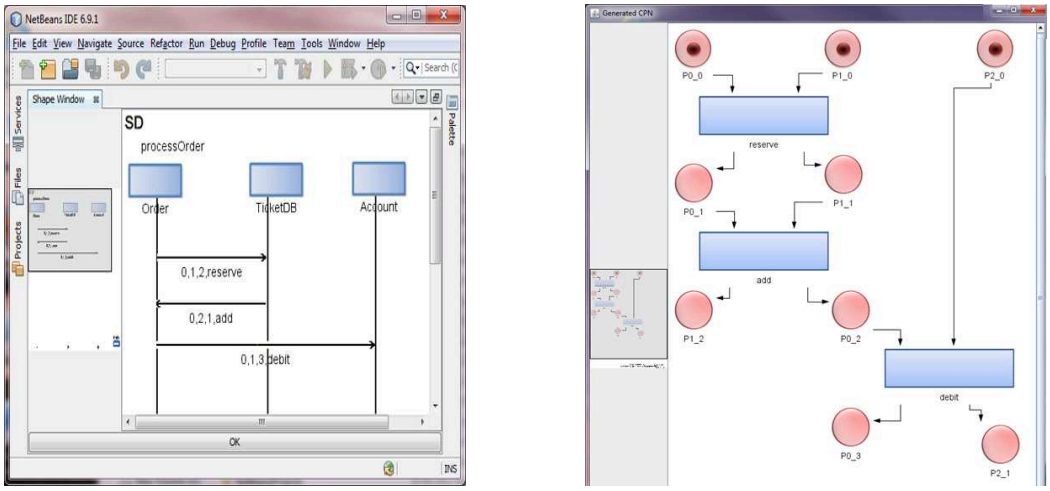
The class *MyConnectionProvider* implements *ConnectProvider*, which is a NetBeans Visual API interface to control a connect action. It checks whether

a specified source or target widget is possible for source or target connection, respectively and creates a connection between the specified source and target widget.

8.3 SD2CPN in Operation

A simple scenario is explained below, which represents the runtime behaviour of the implemented tool.

Consider a simple scenario of an order processing system, which facilitates ticket reservation. When the ticket is reserved it adds to the order and debits from the account. For this interaction, the corresponding SD consists of three instances named *Order*, *TicketDB* and *Account*. First, the instance *Order* sends a local transition with a message label *reserve* to the instance *TicketDB*. Then the class *TicketDB* sends the reply to the class *Order* with the message label *add*. Finally, the class *Order* sends a local transition with a message label *debit* to the class *Account*.



(a) (b)

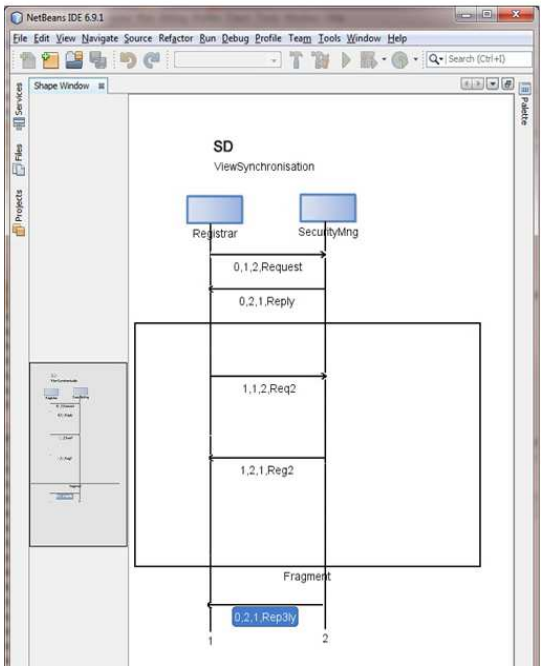
Figure 8.9: A SD with three instances (a) and the corresponding CPN (b) obtained from the SD2CPN tool.

Figure 8.9(a) shows a captured screenshot of a SD with three instances and their interactions drawn using the drag and drop facilities given by the palette GUI. The labels of the diagram elements are typed by the user. Here, the label of a local transition contains four parameters: the first parameter indicates whether the transition is inside an interaction fragment or not, the second and third parameters indicate the identifier of the sending and receiving instances, respectively, and the fourth parameter designates the message label. Once the SD is completed user has to press the OK button at the bottom of the GUI and the tool generates the corresponding formalism for the input SD. Figure 8.9(b) shows the corresponding CPN representation given by automated model transformations implemented within the tool. The CPN shows the places of each colour that correspond to the instances in the SD, net transition and the arcs that link the places and net transitions. Also, it represents the tokens associated with the places.

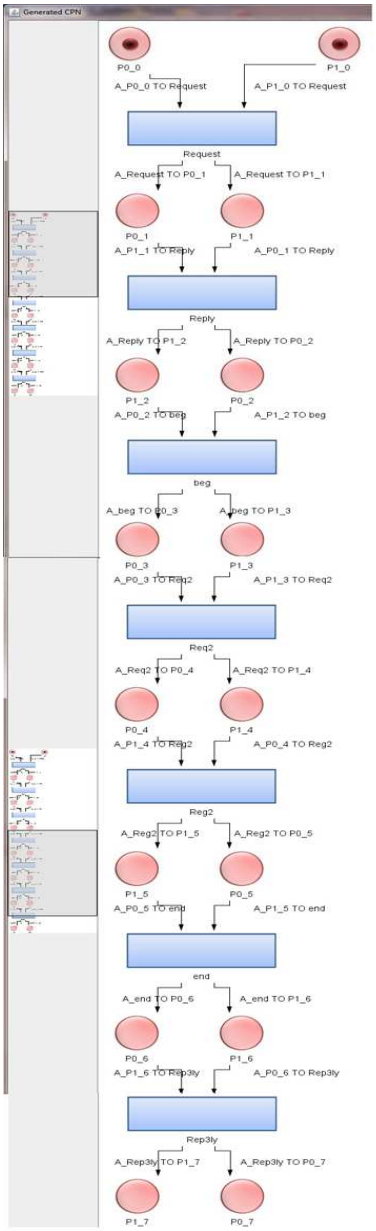
A further complex scenario with an interaction fragment was used and is explained below, in order to examine the capabilities of the tool to handle complex transformations. Consider a SD with two instances named $Registrar$ and $SecurityMng$, and with an interaction fragment that synchronises the interactions at the beginning and end of the fragment. Let the interactions start with two local transitions communicate between $Registrar$ and $SecurityMng$, followed by an interaction fragment. There are two local transitions within the fragment and another local transition after the fragment. Figure 8.10(a) illustrates a SD with a general interaction fragment and Figure 8.10(b) shows the corresponding CPN with the synchronisation behaviour.

In the CPN the net transitions *beg* and *end* correspond to the beginning and the end of the interaction fragment. The obtained CPN representations are intended to support in validating the behaviour given by the SDs; hence

the shows the possibility of automating the defined transformation rules.



(a)



(b)

Figure 8.10: A SD with an interaction fragment (a) and the corresponding CPN with the synchronisation behaviour (b) obtained from the SD2CPN tool.

8.4 SD2CPN Tool with Textual Support

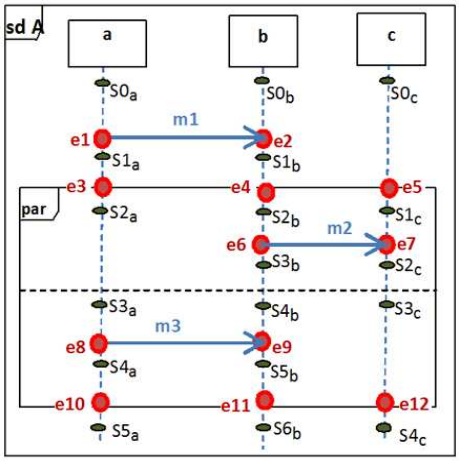
The SD2CPN tool is facilitated with a text-based input and output for integrating the automated model transformations with the existing tools. The grammar for the input SD and the output CPN models are given based on $Backus - Naur\ Form$(BNF). BNF is a notation technique that can be used to describe the syntax of a modelling language. Although many BNF recommendations are available in the literature [Reniers, 1998], we have considered the symbols that are relevant to define our textual grammar for the models.

The textual grammar for a model representation consists of a header, a body and an end. The header consists of the name of the diagram and the body contains a set of statements. A BNF specification is a set of derivation rules written as `<symbol> := _expression_`, where `<symbol>` is a non-terminal. The non-terminals are indicated in between `<` and `>`; while terminals are considered as keywords. The symbols of the left are replaced with the expression on the right and denoted by the symbol `:=`. In general, statements with terminals and non-terminals denote concatenation. Here, `_expression_` consists of one or more sequence of symbols and more sequences are separated by a | that indicates a choice.

This BNF-based grammar uses the symbols `[ ]`, `{ }` to represent optional and grouping statements, respectively. Further, the symbols |, `*`, `+`, `""` indicate alternative, repetition for zero or more, repetition for at least once, and empty string, respectively. The textual representations described in this section are reduced for the purpose of a concise description of the syntax and semantics definitions of the models. The textual representation of the design models are intended for exchanging these models between computer tools only. Following sections explain the model representation using BNF with examples in more detail.

8.4.1 Text Grammar for a Sequence Diagram

This section defines a BNF-based grammar for a sequence diagram that conforms to the SD formal definitions given in Chapter 3, Definition 3.1. The textual representation of a SD enables further formal processing on the model. Here, we focus on the $event - oriented$ description for the explanation of the textual syntax. The list of events corresponds to the order as they are expected to occur in a trace of the system or as come across while scanning the diagram from top to bottom. The $event - oriented$ syntax over $instance - oriented$ form is chosen for many reasons [Reniers, 1998].



(a)

```
SD A begin:

instance a: a : S0a
instance b: b : S0b
instance c: c : S0c

transition : a,m1,b : S1a,S1b

Beg Frag 1: par
Beg Op 1:1; a:S2a, b:S2b, c:S1c
transition : b,m2,c : S3b,S2c
End Op: 1:1
Beg Op 2:1; a:S3a, b:S4b, c:S3c
transition : a,m3,b : S4a,S5b
End Op: 2:1
End Frag 1; a:S5a, b:S6b, c:S4c

SD end.
```

(b)

Figure 8.11: A SD with a parallel behaviour (a) and the corresponding textual representation (b).

Mainly the instance-oriented representation requires many redundant data. For example, the data relevant to a local transition have to be described for each instance that they are defined. However, in event-oriented form it is possible to describe a local transition once for all instances involved. Also, this form facilitates to represent the local transition execution order while respecting the event ordering of each instance.

<Sequence Diagram>:= <SD head> <SD body> <SD end>

<SD head>:= SD <SD name> begin:
<SD body>:= <Instance declaration> [<Gate Statement>] <SD statement>*
<SD end>:= SD end.

<SD name>:= String

<Instance declaration>:= {<Instance statement> | <Lifeline Decomposition statement>} : <Initial state location statement>
<Instance statement>:= instance <Instance identifier> : <Instance name>
<Lifeline Decomposition statement>:= instance <Instance identifier>: <Instance name> (ref <SD name>)
<Instance identifier>:= Integer/ char
<Instance name>:= String

<Initial state location statement>:= <state location statement>
<state location statement> := {S<Local order> <Instance identifier>} | {Se<Local order>}
<Local order>:= Integer

<Gate Statement>:= gate <Gate Identifier> : <Initial state location statement>
<Gate identifier> := String

<SD statement>:= <Local transition statement> | <Interaction fragment statement > | <Interaction reference statement> |<New instance statement>

<New instance statement>:= <Instance statement>: <Initial state location statement>

<Local transition statement>:= transition: {<Sender identifier>, <Message label>, <Receiver identifier>} : <Sender state location statement>, <Receiver state location statement>

<Sender identifier>:= <Instance identifier> | <Gate identifier>
<Message label>:= String
<Receiver identifier>:= <Instance identifier> | <Gate identifier>
<Sender state location statement>:= <state location statement>
<Receiver state location statement>:= <state location statement>

Figure 8.12: The text grammar for a sequence diagram.

```
<Interaction fragment statement>:= Beg Frag <Fragment identifier> : <Fragment type>
                                    <Operand statement>*  <End Frag statement>
<Fragment identifier>: Integer | character

<Fragment type>:= alt | loop | par | break | option | seq | strict | assert | neg | consider |
                  ignore | critical | ref <SD name>

<List instances>:= <Instance identifier> : <state location statement>

<Operand statement>:= Beg Op <Operand identifier> [: <Conditional statement>];
                      <List instances>+  <SD statement>+  <End Op statement>

<Operand identifier>:= integer : <Fragment identifier>
<Conditional statement>:= <Instance identifier>
                       {[<Variable> <Operator> <Variable value>]}
<Variable>:= String
<Operator>:= < | > | {<=} | {>=} | {==} | {!=}
<Variable value>:= Integer

<End Op statement>:= End Op : <Operand identifier>: Fragment identifier>
<End Frag statement>:= End Frag <Fragment identifier>; <List instances>+

<Interaction reference statement>:= Beg Frag <Fragment identifier> : <Fragment type>;
                                    <List instances>+ <Local transition statement>
                                     <End Frag statement>
```

Figure 8.13: The text grammar for a sequence diagram cont.

Figure 8.11 shows an example of a SD with parallel behaviour and the corresponding text-based representation that complies with the SD text grammar shown in Figure 8.12 and Figure 8.13.

The textual grammar of the SD consists of a `<SD head>`, `<SD body>` and `<SD end>` (see Figure 8.12). The header consists of the name of the diagram, preceded by a keyword `SD` and followed by a keyword `begin:`. The SD body contains a set of `<Instance declaration>`, `<Gate statement>` if available and a set of `<SD statement>` that describe the interactions within

the diagram.

The declaration of an instance consists of either an instance or a lifeline decomposition statement followed by the initial state location statement of that instance. The instance statements state the existence of the object instances with their identifier and the name. The identifier uniquely distinguishes an instance and used as a reference to the instance throughout the interactions of the diagram. The statement `<Initial state location statement>` is same as a `<state location statement>` and it is textually described using a symbol `S`, the local order and the instance identifier of that state location.

When a SD becomes complex, gates are used as an interface between the considered diagram and the environment. In such situations, the local transitions that are sent to and received from the environment are indicated by the gate identifier, instead of an instance identifier. `<Gate Statement>` states the existence of the environment instances with the keyword *gate* followed by the identifier and the associated initial state location separated by the symbol `:`.

Another method to resolve the complexity of a SD and support different views of abstraction is lifeline decomposition. `<Lifeline Decomposition statement>` includes in the `<Instance declaration>` statement and specifies the instance statement with the `ref` keyword followed by the referred SD name.

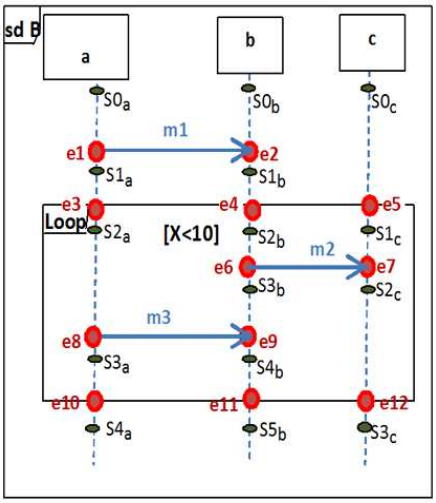
The statement `<SD statement>` describes an occurrences of a local transition that leads to interactions, an interaction fragment, a reference behaviour, or a new instance creations within the diagram. Textually a local transition is described using a keyword `transition:` followed by a set of identifiers for the sending instance, message label and the receiving instance, respectively. Here, the sender and the receiver identifiers can be an `<Instance identifier>` or a `<Gate identifier>` type. Further, this statement is followed by the state

location statements of the sender and the receiver of the local transition. This textual syntax helps to distinguish between multiple occurrences of the same message name.

The text grammar for the behaviour of an interaction fragment is continued in Figure 8.13. The statement `<Interaction fragment statement>` starts with the keyword `Beg Frag` followed by its identifier, the type (*alt*, *par*, *etc.*), the associated statements for the operands (`<Operand statement>`) and `<End Frag statement>`. Each of `<Operand statement>` begins with the keyword `Beg Op`, an identifier, an optional conditional statement, involved instance identifiers and the associated state locations given by the *min* function. A conditional statement is included when the fragment type is `alt`, `loop`, `option`, `break` that restrict the possible continuation of interactions. The textual grammar for a conditional statement contains the associated instance identifier, which executes the condition and the conditional expression with two variable values and an operator.

Further, an operand statement contains a set of `<SD statements>` that specifies the behaviour within the operand and ends with `<End Op statement>`. The statement `<End Op statement>`, contains the keyword `End Op` followed by the corresponding operand identifier and the associated fragment identifier. Further, `<End Frag statement>` consist of the keyword `End Frag` followed its identifier, involved instances and the state locations given by the θ function.

The textual grammar given in this section describes the representation of the elements of the SD and does not describe the behaviour of each fragment associated with the diagram. This textual grammar includes an extension for the behaviour of instance creation and can be easily extended for the all other behaviours in a SD such as instance destruction, lost and found messages, time and stochastic annotations.




```
SD B begin:

instance a: a : S0a
instance b: b : S0b
instance c: c : S0c

transition: a,m1,b : S1a,S1b

Beg Frag 1: loop
Beg Op 1: 1 [x<10]; a:S2a, b:S2b, c:S1c
transition: b,m2,c : S3b,S2c
transition: a,m3,b : S3a,S4b
End Op: 1:1
End Frag 1; a:S4a, b:S5b, c:S3c

SD end.
```

(a) (b)

Figure 8.14: A SD with an iterative behaviour (a) and the corresponding textual representation (b).

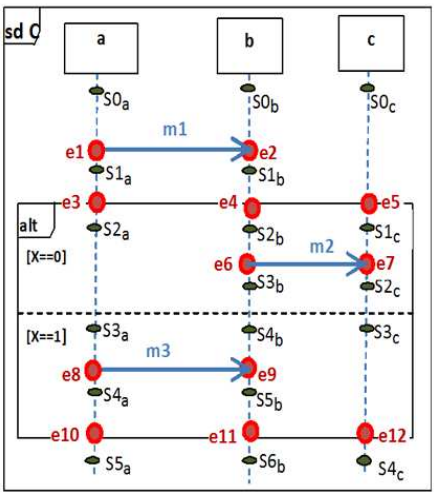
Consider the examples with the SDs *sd B*, *sd C* and *sd D* and the corresponding text grammar shown in Figure 8.14, Figure 8.15 and Figure 8.16 that represent SDs with iterative behaviour, alternative behaviour and reference behaviour with gate events, respectively.

The diagram *sd B* consists of three instances a, b and c. The corresponding textual representation declares an instance with its identifier, name and initial state location such that `instance 1: a : S0a`. The textual representation for the first local transition, `transition: a,m1,b : S1a,S1b`, specifies that the transition with the message label $m1$ is sent from the instance a to b and the corresponding state locations are $S1a$ and $S1b$.

The beginning of the fragment is represented by `Beg Frag 1: loop` that indicates the fragment identifier and the type. The *loop* interaction fragment contains only one operand and associates with a constraint that specifies the loop condition. This is specified as `Beg Op 1: 1 [x<10];` and fol-

lowed by the instance identifiers and minimum state locations involved in the operand such that $a : S2a,\ b : S2b,\ c : S1c$.

After specifying the interaction within the operand, the end statement of the operand is specified with the operand identifier and the fragment identifier such that `End Op: 1: 1`. When the end of the fragment is reached, the statement `End Frag 1; a:S4a, b:S5b, c:S3c` specifies the fragment end with its identifier, followed by the involved instances and the associated state locations after the end of the fragment. Finally the keyword `SD end.` represents the end of the diagram.

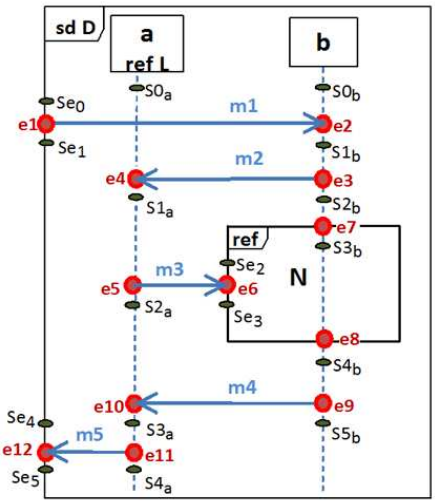



```
SD C begin:

instance a: a : S0a
instance b: b : S0b
instance c: c : S0c

transition: a,m1,b : S1a,S1b

Beg Frag 1: alt
Beg Op 1: 1 [x==0]; a:S2a, b:S2b, c:S1c
transition: b,m2,c : S3b,S2c
End Op: 1:1
Beg Op 2: 1 [x==1]; a:S3a, b:S4b, c:S3c
transition: a,m3,b : S4a,S5b
End Op: 2:1
End Frag 1; a:S5a, b:S6b, c:S4c

SD end.
```

(a) (b)

Figure 8.15: A SD with a parallel behaviour (a) and the corresponding textual representation (b).

Similarly, the textual representation of the diagram *sd C* shown in Figure 8.15 shows the fragment representation with two operands, where each operand contains a conditional statement. Here, the second operand starts after the end of first operand and the fragment end reaches after `End Op: 2:1` that represents the end of operand 2 in fragment with the identifier 1.



(a)

```
SD D begin:

instance a: a (ref L) : S0a
instance b: b : S0b

gate g1 : Se0
gate g2 : Se2
gate g3 : Se4

transition: g1,m1,b : Se1,S1b
transition: b,m2,a : S2b,S1a

Beg Frag 1: ref N; b:S3b
transition: a,m3,g2 : S2a,Se3
End Frag 1; b:S4b

transition: b,m4,a : S5b,S3a
transition: a,m5,g3 : S4a,Se5

SD end.
```

(b)

Figure 8.16: A SD with gate elements and reference behaviour (a) and the corresponding textual representation (b)

The diagram *sd D* in Figure 8.16 shows more complex behaviour of a SD with lifeline decomposition, reference behaviour and gate events. The textual representation for a gate event is given by the gate identifier with its initial state location that belongs to the environment such that `gate g1 : Se0`. The lifeline decomposition is specified in the instance declaration such that `instance a: a (ref L) : S0a`, where the *ref* keyword is followed by the referred SD. The statement `<Interaction reference statement>` starts with `Beg Frag` and ends with `End Frag` gives the referred diagram name such that *ref N*.

8.4.2 Text Grammar for a CPN

This section defines the text grammar for a CPN that complies with the CPN definition given in Definition 4.1. This event-oriented textual representation focuses the object control flow with the execution order of the CPN model.

```
<CPN>:= <CPN head> <CPN body> <CPN end>
<CPN head>:= CPN < CPN name> begin :
<CPN body>:= <Colour declaration> < CPN statement>*
<CPN end>:= CPN end.

<CPN name>:= String
<Colour declaration>:= {<Colour statement> | <Reference statement>}

<Colour statement>:= colour <Colour identifier> : <Colour name>
<Reference statement>:= colour <Colour identifier> : <Colour name> (ref <CPN name>)
<Colour identifier>:= Integer/ char
<Colour name>:= String

< CPN statement>:= <Place statement> | <Net-Transition statement> | <Arc statement>

<Place statement>:= place: <Place name>  [:<NoOfTokens>]
 <Place name>:=  P<Local order><Colour identifier>
<Local order>:= Integer
<NoOfTokens>]:= Integer

<Net-Transition statement>:= transition: <Transition name> [<Conditional statement>]
<Transition name>:= <Transition identifier>:<Transition label>
<Transition identifier> :< Integer>
<Transition label>:= String

<Arc statement>:= arc: A_<Place name> TO <Transition name>}| <Transition name>}
                                TO <Place name> } [<Conditional statement>]

 <Conditional statement>:= {<Variable><Operator><Variable value>}
<Variable> := String
<Operator>:= {< | > | <= | >= | == | !=}
<Variable value>:= Integer
```

Figure 8.17: The text grammar for a CPN.

The text grammar of the CPN is represented based on BNF and includes a header, a body and an end (see Figure 8.17). The header is specified using the keyword `CPN` followed by the diagram name and the keyword `begin:`. The body of the text grammar contains the statements `<Colour declaration>` and `<CPN statement>`. The statement `<Colour declaration>` consists with

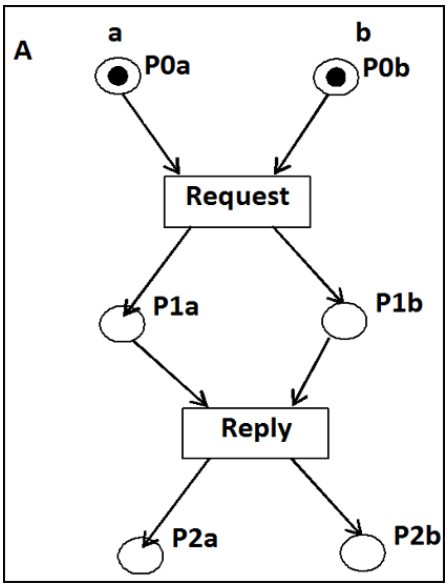
either `<Colour statement>` with a keyword *colour*, an identifier and a name, or `<Reference statement>` that specifies a colour identifier, name and the referred diagram name preceded by a keyword `ref`.

A `<CPN statement>` describes the existences of a place or an arc or a net-transitions. The statement `<Place statement>`, consists of the keyword *place* and the name of the place, which is derived from its local order and the colour identifier. Optionally, a statement of a place may contain the number of token associated with it. The statement `<Net-Transition statement>`, is specified with the keyword `transition:` followed by `<Transition name>`, which consists of a transition identifier and the associated label.

If the net transition is associated with a conditional statement, it is specified within the symbols `[ ]`, as an optional element. Here, the textual grammar for a conditional statement contains two variable values and an operator that gives the expression. To specify the link between the places and net transitions, `<Arc statement>` is used. The statement `<Arc statement>`, is described with the keyword `arc:` followed by `A_<source> To <target>`, where `<source>` and `<target>` is a `<Place name>` and a `<Transition name>` statement, respectively or vice-versa. Here, the letter `A` indicates an arc.

Since this text representation of a CPN is generated by transforming a SD, some of the element identifiers in the CPN text grammar are same as the identifiers in the SD grammar. For example, the name of the CPN corresponds to the name of the SD, the statement of a colour corresponds to a statement of an instance. Further, the label of a transition corresponds to a message label or to a fragment name followed by the keywords `beg` or `end`.

Consider the graphical and the textual representations of a CPN shown in Figure 8.18. The CPN consists of two colours a, b, two net transitions and the associated places and arcs. The textual statement `place: P0a :1` indi-




```
CPN A begin:

colour a: a
colour b: b

place: P0a :1
place: P0b :1
arc: A_P0a To 1:Request
arc: A_P0b To 1:Request
transition: 1:Request
arc: A_1:Request To P1a
arc: A_1:Request To P1b
place: P1a
place: P1b
arc: A_P1a To 2:Reply
arc: A_P1b To 2:Reply
transition: 2:Reply
arc: A_2:Reply To P2a
arc: A_2:Reply To P2b
place: P2a
place: P2b
CPN end.
```

(a) (b)

Figure 8.18: A CPN (a) and the corresponding textual representation (b).

cates that the place $P0a$ contains one token. The initial places are followed by the arcs that link the initial places with the following net transition with the label *Request*. For example, the text representation `arc: A_P0a To 1: Request` specifies that there is an arc where the source element is the place $P0a$ and the target element is the net transition $1 : Request$. The first transition is textually represented by `transition: 1:Request` where *Request* is the associated label of the transition. Based on the execution order the remaining statements are listed and finally the end of the model is specified using `CPN end.`.

8.5 SD2CPN Tool Implementation

This section describes the implementation procedure of the SD2CPN tool. The input model for the tool can be given in either graphical or textual format and the output model can be generated in both graphical or textual notations (Fig-

ure 8.19). Also, the graphical input can be used to generate the corresponding textual notations as described in Section 8.4.

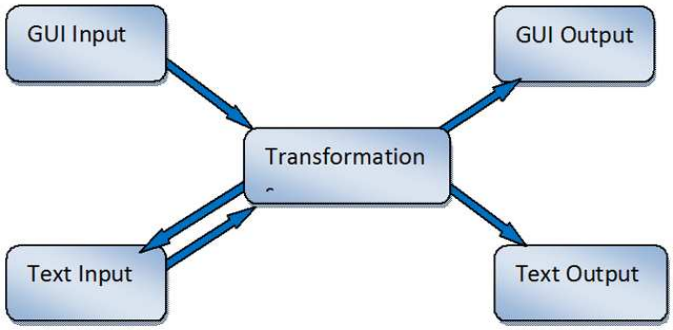


Figure 8.19: The input output representations of the SD2CPN tool.

The component *SD Presenter* in the component *View Component* (Figure 8.2) inputs the elements and their relationships in a SD using the GUI or the textual notations in a format that can be easily processed by the component *Process Component* of the tool. When the input is given graphically, the user enters the label of the message element with four parameters; the first parameter specifies a reference for a fragment identifier to indicate whether the local transition is within a fragment or not. The second and third parameters specify the identifiers for the send and receive instances and the final parameter indicates the message name.

Figure 8.20 and Figure 8.21 illustrate the execution of the component *Process Component* during the transformation of a SD to a CPN considering the basic and complex elements, respectively. The process starts by getting the input data of the SD and identifying the instances of elements and their associations correspond to the SD meta-model. As the first step towards transformations, an object of a SD model is created and the corresponding object of the CPN is created with the same diagram name. Then it process an element by element and while building the formal SD model, it transforms each

SD element to the corresponding CPN element; thus builds the CPN formal model.

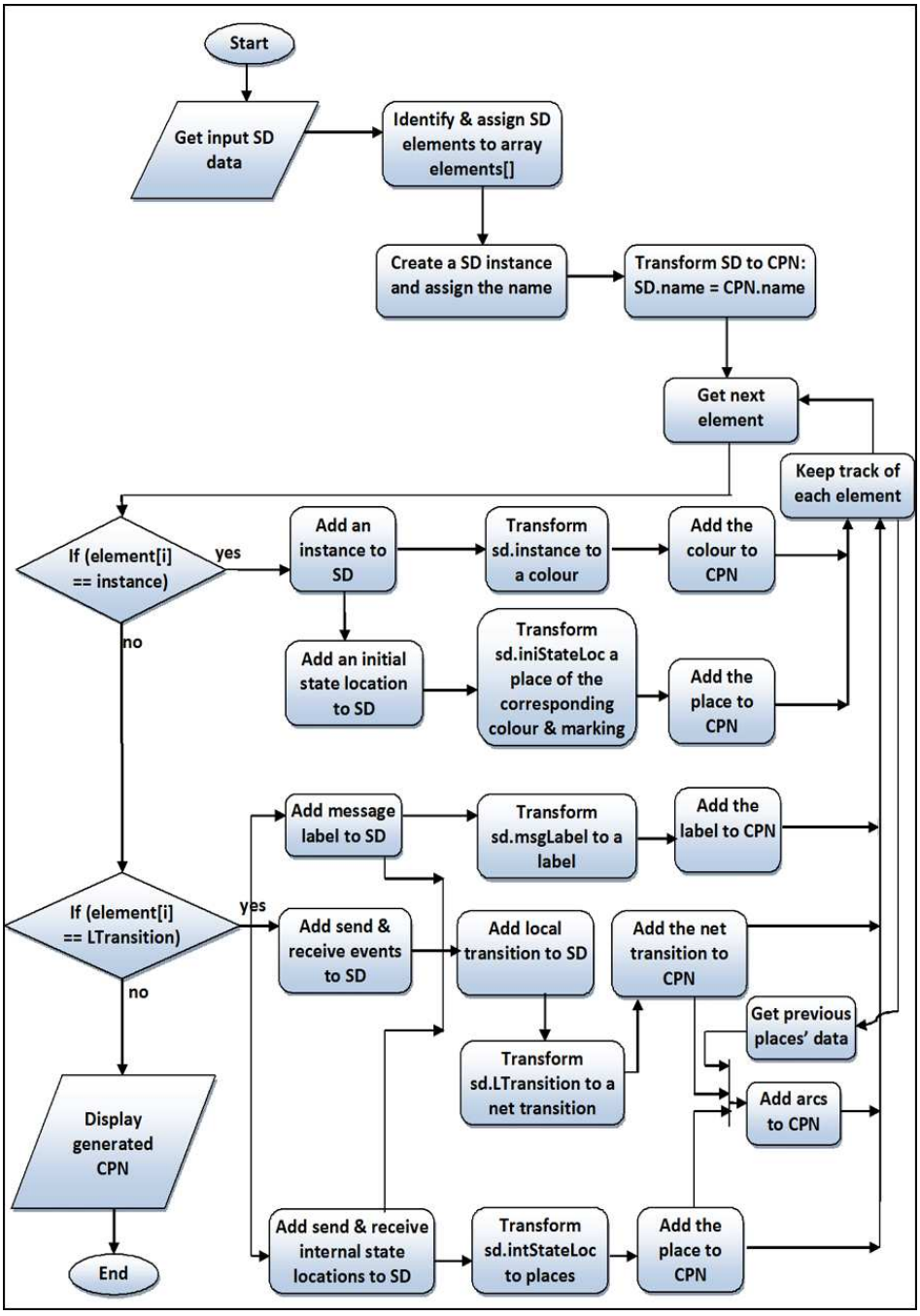


Figure 8.20: The flow chart for the basic transformations of the SD2CPN tool.

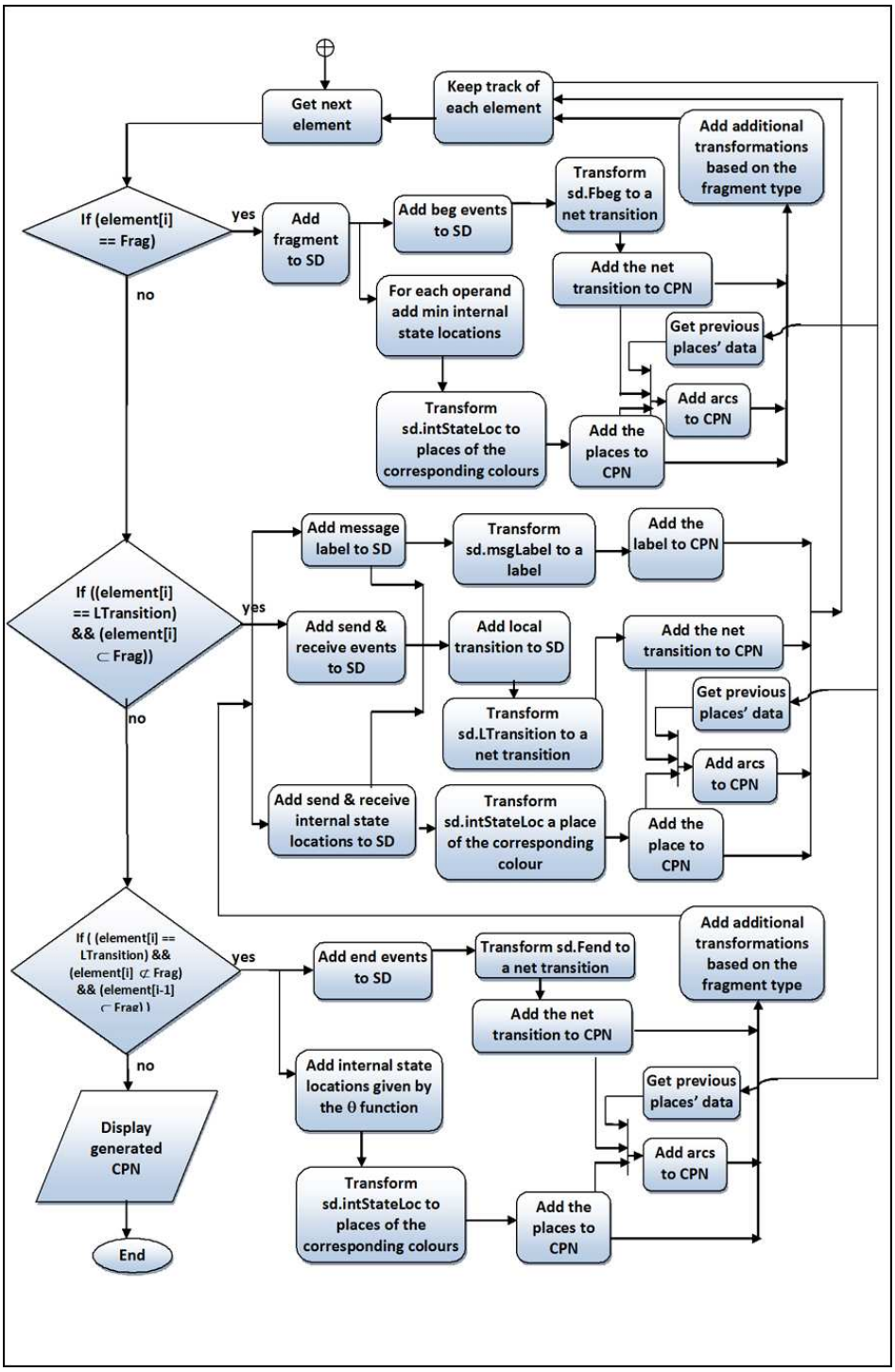


Figure 8.21: The flow chart for the complex transformations of the SD2CPN tool.

For example, consider the flow chart given in Figure 8.20. If the element is an instance, it is added as an class *Instance* of the SD meta-model, transformed it to a class *Colour* of the CPN meta-model and added as an object of the class *Colour* of the CPN model. The process also adds the initial state location associated with the instance. This object *StateLocation* is transformed to an object *place* with the corresponding colour and marking and added to the CPN model.

If the element is a local transition that consists of a message label and two events, the process starts by adding the corresponding object of the class *messageLabel* and the objects of the associated classes *Event* and *StateLocation* to the SD model (which is an object of the class *SD*). Then it adds the object of the class *LocalTransition* to *SD*. After that each of these objects; i.e. message label, local transition and state locations are transformed to the objects of the classes *Label*, *NetTransition* and the *Place*, respectively, and added as the objects of the CPN model. The process retrieves the input places of the net transition and adds the corresponding objects of the class *Arc* that link the newly added net transition and the associated places. Next, the process updates the status of each object array that used to keep the flow control and retrieves the next object.

The process given in Figure 8.21 is a continuation with an interaction fragment behaviour. Here, when the element is a fragment the process adds an object of the class *InteractionFragment* with the operands. It also adds the associated objects of the class *Event* for the beginning of the fragment and the objects of the class *StateLocation* of each operand that are given by the *min* function. The beginning of the fragment is transformed to an object of the class *NetTransition* while the created object of the class *StateLocation* are transformed to corresponding object of the class *Place* and added to the CPN.

By linking with the previous places (source place of an arc), the objects of the class *Arc* are created between the net transition and the associated places, and added to the CPN. Additionally, the transformations specific to the fragment type, such as imposing equality between some given places, are performed.

After the processing of the local transitions within a fragment, and when it encounters a local transition outside the fragment, the end of the fragment is processed. Here, the objects of the class *Event* are associated with the end of the fragment and the objects of the class *StateLocation* after the fragment are added to the SD model and transformed into the corresponding objects of the classes $NetTransition$ and *Place*, respectively. Then, the objects of the class *Arc* are added to the CPN model with the use of previous places data. Additionally, the guard expressions and associated variables can be incorporated as user inputs to the SD and process as extra tasks. With the transformations from a SD to a CPN, the number of net transition in the CPN = number of local transitions + (2 $\times$ number of fragments). When all the SD elements are transformed to the corresponding element of the CPN, it displays the generated CPN by calling the process within the component $View\ Component$.

The algorithm of a transformation rule can be specified as follows: each rule starts with the keyword $Transformation$ and a name that specifies the source and the target elements. The source and the target languages are referenced by stating the both model names between brackets, where the first name indicates the source model and the second indicates the target model, following the transformation name. The naming complies with the standard programming qualifiers. The element declarations and the mapping rules are written within the curly brackets. The mapping rules used in the implementation are conceptually referred to the transformation rules defined in Chapter

5. Further, complex transformation rules can be constructed using the basic mapping rules.

Here, the elements of the source and the target models are written as variable declarations following the keywords *source* and *target*, respectively. The type of the element is of the type defined in the corresponding meta-model. The parameters used for the transformation process are listed following the keyword `params`. The mapping rules start with the keyword *mapping*. The mapping rules are specified using the infix operator ($\sim$) with two operands and specifies that the transformation will map the operand in the LHS to the operand in the RHS. Further, in some situations the operand may denote a set of elements. The details of the mapping rule, such as the equality between the properties of the elements are listed as given by the transformation rules in Chapter 5.

e.g. 8.1 *Transformation of a state location to a place=2*

```
Transformation StatelocationToPlace (SD, CPN) {
source: StateLocation s;
target: Place p;
params: null;
mapping:  s  ~  p;
    p.name = s.name;
    p.colour = s.instance;
 }
```

Consider the algorithm given in Example 8.1. It states that for each state location there there is a transformation that maps that state location to a place. This mapping rule is conformed to Rule 5.3 in Chapter 5. Here, the source model is a SD and the target model is a CPN. The source element is a

state location and the target element is a place. This transformation does not need any additional parameters and the mapping is from a state location to a place, in such a way that the name and the instance of the state location are mapped to the name and the colour of the place, respectively.

e.g. 8.2 *Transformation of a local transition to a net transition*

```
Transformation LocalTransitionToNetTransition (SD, CPN) {
source: LocalTransition lt;
target: NetTransition nt;
params: prePlace[], postPlace[];
mapping:  lt  ~  nt;
    nt.label = lt.msgLabel;
    foreach prePlace[i] && postPlace[i]
   nt.addArc(new Arc(prePlace[i],this));
   nt.addArc(new Arc(this,postPlace[i]));
 }
```

The algorithm given in Example 8.2 shows the transformation of a local transition to a net transition as defined in Rule 5.5. The places that should be associated with the net transition are passed as parameters of this transformation. This algorithm maps a local transition to a net transition by assigning the message label to the net transition label. Additionally, it creates arcs that link the given places with the net transition. When the net transition links with the previous and post places, the target element of the arc becomes the net transition and the place, respectively.

Figure 8.22 shows the implementation tasks for the graphical representation of the component *CPN Presenter* in *View Ccomponent*. Here, the procedure takes the elements of the generated CPN and draws the corresponding icons

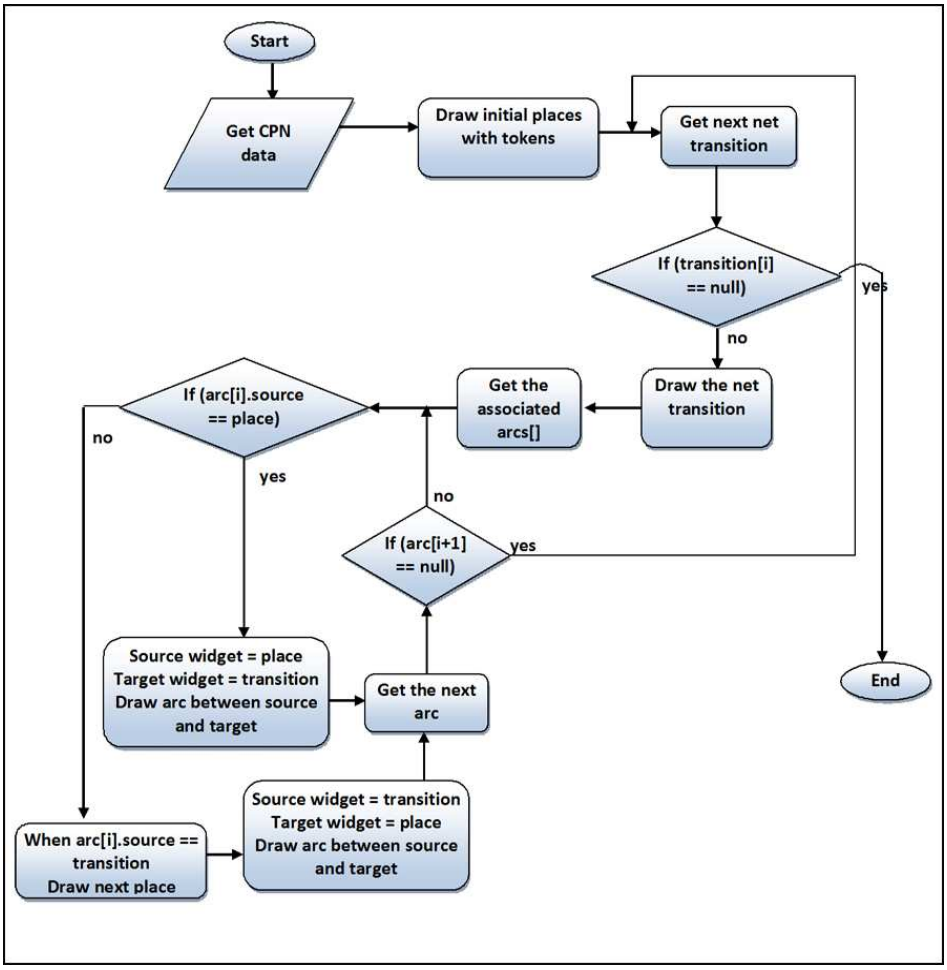


Figure 8.22: The flow chart for the graphical representation of the CPN Presenter.

for the initial places with tokens. Then for each net transition, first it draws the icon for the net transition and retrieves the associated arcs. For each associated arc, if the source of the arc is a place then it connects the source place and the target net transition using an arc. Otherwise, in the case where the source of the arc is a net transition, the tool draws the relevant place and then draws an arc connecting the source net transition and the target place. Similarly, the textual representation the CPN can be generated.

8.6 Model Transformation using Case-Studies

Generally, the applicability of a model transformation defined as a set of transformation rules can be best investigated through the analysis of case studies and examples. We have shown that our model transformation as defined is semantically correct, and in this section we investigate its applicability and usability in practice.

We validate the applicability of our proposed model transformation using two different software system examples as case studies. This also helps to evaluate the practical usefulness of the defined transformation rules. These examples cover the different levels of system functions and contexts giving a complete coverage for the expected analysis of the proposed formal transformations.

The first case study considers how a cloud computing service provision operates, and shows the applicability of the proposed framework in an increasingly popular domain with many applications. Not only does the case study shows the transformations within the cloud system context, but also illustrates the applicability of the rules in the business process. The second case study is based on an abstract specification of an elevator [1] system which we use as an example to see how usual operational conditions are transformed from SD to CPNs. The importance of this example to the thesis is that it gives an insight into a more refined everyday standalone system, as opposed to the first case study on high-level virtual services and interactions. Therefore, both case studies cover different levels of detail, system specification and context, giving a more complete coverage for the usability and expected analysis of the proposed formal transformation.

[1] The word Elevator (Engish-US) is used instead of Lift (English-GB) for clarity

8.6.1 Example 1: Cloud Service System

The cloud service based example described in this section illustrates the parametric transformation technique and the associated transformation rules defined in Chapter 6. The example contains a scenario from cloud computing with timing and stochastic behaviour of interest, which can be analysed separately. A simplified form of this example was given in [Bowles and Meedeniya, 2012a].

Cloud computing is a new paradigm for the dynamic provision of on-demand computing services that uses the Internet as a platform to share resources [Buyya et al., 2009]. Here we consider an example that explains an on-demand service hosting and service management environment of a cloud.

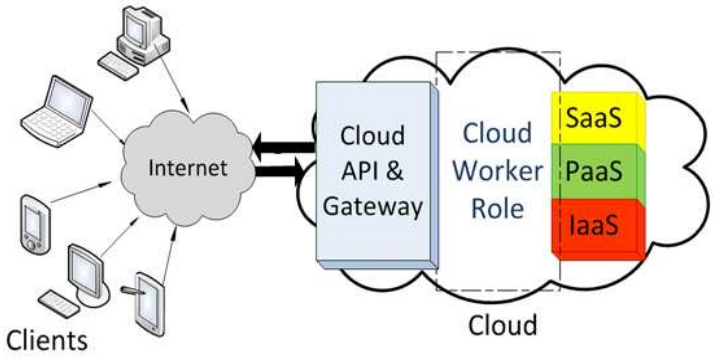


Figure 8.23: An overview of a Cloud Computing System.

Figure 8.23 shows an abstract view of a cloud environment.

- *Client*: is a client (from a group of users) instance that access the cloud service;

- *Cloud API & Gateway*: is a virtual instance, which accepts the incoming HTTP requests from the *Clients*. Depending on the implementation of the system architecture, *Cloud API & Gateway* can be located partially

inside or outside the cloud. Different clients can initiate their own requests for different services and the *Cloud API & Gateway* handles these requests;

- *Coloud Worker Role*: is a working instance of the cloud resource provision.

Mainly, a cloud system delivers three types of services: (1) Infrastructure as a Service (IaaS) that allocates resources such as CPU power, storage, network for computation, etc., (2) Platform as a Service (PaaS) that virtualises a hardware infrastructure such as OS, application engines, etc., and (3) Software as a Service (SaaS) that provides utility applications for the clients. These services are made available as subscription-based services in a pay-as-you-go model to consumers. This example focuses on resource management in SaaS.

Here, *Cloud Worker Role* retrieves the job from *Cloud API & Gateway* and acquires necessary resources based on the client's subscription type: *single-tenancy* or *multi-tenancy*. If the subscription type is single-tenancy (a more secure option) then the dedicated resources are obtained, otherwise shared resources are used. *Cloud Worker Role* calls outside services to satisfy the request and once the completed result is received, this is sent to *Client* via *Cloud API & Gateway*. In this way, *Cloud API & Gateway* facilitates scalability and client specific state management. Hence, a client can send multiple requests to the same *Cloud API & Gateway*. Here, the session data can be internally handled to keep track of the status of each task, and to notify when a task is completed.

We are interested in the behaviour of a scenario concerning a high-level service request from a cloud system. The model is inspired by a client service request from a cloud system and the necessary resource management services. The scenario is described as a set of high-level functional synchronisations

between *Client*, *Cloud API & Gateway* and *Cloud Worker Role*. It is assumed that the client has already authenticated.

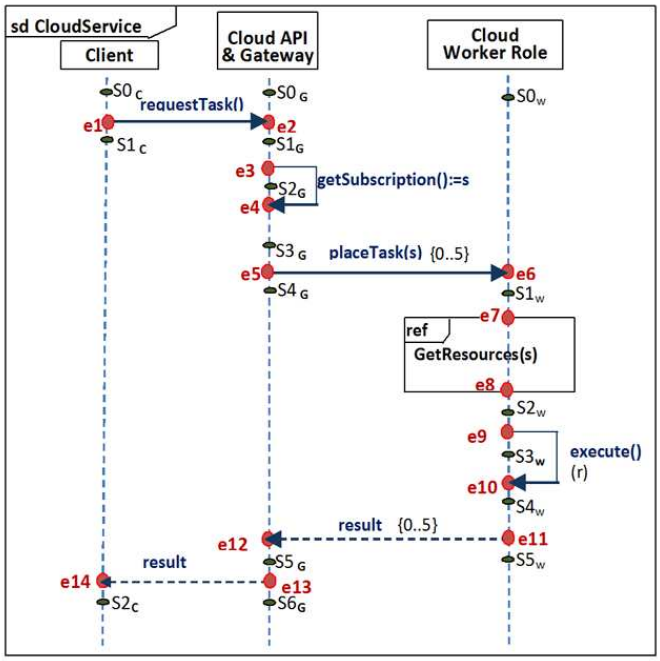


Figure 8.24: A Cloud System sequence diagram.

Consider the sequence diagram shown in Figure 8.24 with three instances where events and state locations along instance lifelines are indicated explicitly. The interaction $SD_{CloudService}$ is initiated by the instance *Client* sending a local transition with the message label *requestTask()* to the instance *Cloud API & Gateway*. *Cloud API & Gateway* then executes a self-transition to get subscription information for the client (with return value s) and places a new task on the instance *Cloud Worker Role*.

This interaction is followed by a ref interaction fragment that illustrates the reference behaviour. The instance of *Cloud Worker Role* gets the required resources by referring another sequence diagram named *GetResources(s)*, which executes the request and returns the result which is subsequently forwarded

to the client.

In this high-level design, the client's subscription type (given by value s) is passed explicitly to the instance $SD_{GetResources}$ as an argument and used internally as needed (cf. Figure 8.25). Here, based on the tenancy type, the instance of *Cloud Worker Role* requests resources from a *Dedicated Resource* or a *Shared Resource*. An *alt* interaction fragment is used to select between single-tenancy (ST) or multi-tenancy (MT) choices depending on the value of s. That is, $guard(t_1) = [ST == T]$ and $guard(t_2) = [MT == T]$ where $t_1 = (e_4, RequestDR, e_5)$, $t_2 = (e_6, RequestSR, e_7)$ for $t_1, t_2 \in T_n$.

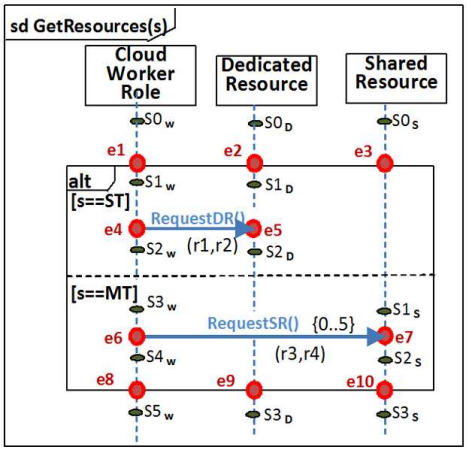


Figure 8.25: The GetResource(s) sequence diagram.

The timing and stochastic information associated with the sequence diagrams in this example are as follows. In the sequence diagram $SD_{CloudService}$, the time constraint on the occurrence of *placeTask(s)* indicates that a task has to be committed to the instance of *Cloud Worker Role* within 5 time units (in case of multiple client requests this may not be possible). Formally this is given by $time_{SD_{CloudService}}(e_5, e_6) = [0, 5]$. Similarly, the time annotation associated with the local transition $t = (e_{11}, result, e_{12})$ is given by $time_{SD_{CloudService}}(e_{11}, e_{12}) = [0, 5]$. Furthermore, message *execute()* has rate r

and we have the stochastic annotation $\mathcal{S} = \{(execute, (r))\}$.

The diagram $SD_{GetResources}$ shows a timing annotation given formally by $\mathcal{T} = \{(RequestSR, [0, 5])\}$. Further, the stochastic annotations are given by $\mathcal{S} = \{(RequestDR, (r_1, r_2)), (RequestSR, (r_3, r_4))\}$, where $rate_{SD_{GetResources}}(t_1) = (r_1, r_2)$ and $rate_{SD_{GetResources}}(t_2) = (r_3, r_4)$ for the local transitions $t_1 = (e_4, RequestDR, e_5)$ and $t_2 = (e_6, RequestSR, e_7)$.

Applying the basic and interaction fragment transformation rules defined in Chapter 5 and Chapter 6, together with the timing and stochastic annotation Rule 6.9 and Rule 6.10, the corresponding CPN models for this example can be obtained. $CPN_{CloudService}$ in Figure 8.26 and $CPN_{GetResources}$ in Figure 8.27 show the corresponding CPN models for $SD_{CloudService}$ and $SD_{GetResources}$ respectively.

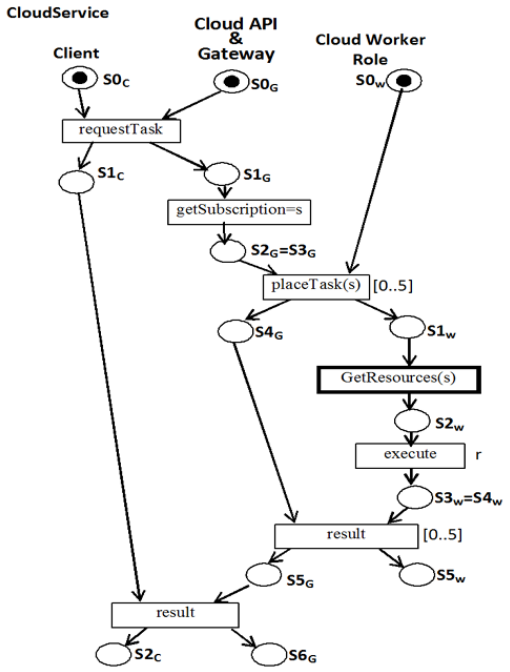


Figure 8.26: A Cloud System CPN Model .

The token colours of the model $CPN_{CloudService}$ correspond to the instances

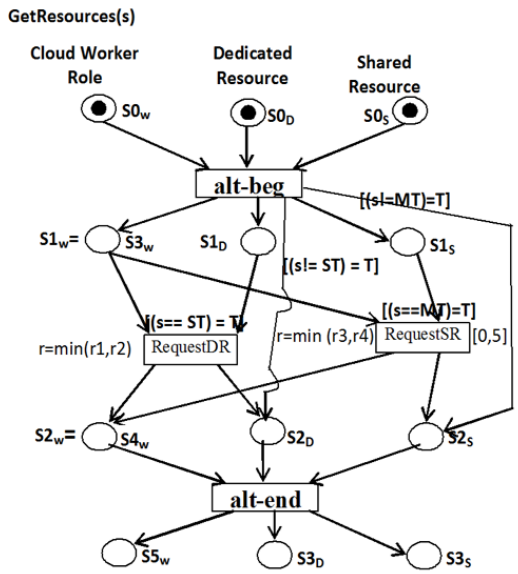


Figure 8.27: GetResource CPN Model .

in $SD_{CloudService}$, i.e., *Client*, *Cloud API & Gateway* and *Cloud Worker Role*. For all the state locations and local transitions in $SD_{CloudService}$ there are corresponding places and net transitions in $CPN_{CloudService}$. Here, matching net transitions are labelled with the message label of the corresponding local transition. For example, $l(t^{'}) = requestTask$ for $t^{'} \in T_n$ and $t^{'} = t$, where $t = (e_1, requestTask, e_2)$. The places and net transitions are linked by arrows such that $node(a) = (S0_C, requestTask)$, and so on, for $a \in A$. The interaction use $GetResources(s)$ is mapped to a net transition $t_r \in T_n$ by applying Rule 6.1, such that $l(t_r) = GetResources(s)$. The timing and stochastic annotations are mapped to the CPN such that $time_{CPN,CloudService}(placeTask) = (0, 5)$, $time_{CPN,CloudService}(result) = (0, 5)$ and $rate_{CPN,CloudService}(execute) = r$.

The model $CPN_{GetResources}$ uses two new net transitions with the labels $alt{-}beg$ and $alt{-}end$ to denote the beginning and the end of the *alt* interaction fragment in $SD_{GetResources}$, and is used in order to synchronise the behaviour before and after the execution of the interaction fragment. In the diagram

$SD_{GetResources}$, since only one operand is selected each time the SD is executed, we optimise the representation of the corresponding CPN model, by having one place to denote the beginning of an arbitrary *alt* operand and one place to denote the end of an arbitrary *alt* operand, for a given instance. The condition associated with each operand in the *alt* fragment are associated with the net transition that corresponds to the first local transition in each operand. For example, $guard(t_{RequestDR}) = [s == ST]$ and $guard(t_{RequestSR}) = [s == MT]$.

The timing and stochastic annotations of $SD_{GetResources}$ are mapped to the corresponding net transitions in $CPN_{GetResources}$ using the functions $time_{CPN}$ and $rate_{CPN}$, respectively. Further, hierarchical transformations can be used to link the models $CPN_{CloudService}$ and $CPN_{GetResources}$ using the model composition Rule 6.6. However this example does not discuss the mapping of hierarchical behaviour, i.e. combining the two diagrams as a one model.

With cloud computing, users are able to access and deploy applications and services from anywhere in the world on demand at competitive costs depending on their QoS (Quality of Service) requirements. To enhance the QoS associated with cloud computing, it is important to measure, for example, the performance of services. The example considered in this section has specified the timed and stochastic data associated with functionalities such as *placeTask()*, *sd GetResources*, and *execute*. The time taken to execute these functionalities and the rate of execution can be analysed separately using existing CPN analysis tools (eg. SimQPN [Kounev and Buchmann, 2006], QPME [Kounev and Dutz, 2007], CPN tools [Jensen and Kristensen, 2009]). For example, applying these tools to the CPN model in Figure 8.27, we can compare the performance of acquiring resources in single-tenancy and multi-tenancy scenarios separately. Further, this real-time and stochastic data can be used to analyse the throughput and the utilisation of performing a service.

8.6.2 Example 2: Elevator System

This section describes an example of a real-time software application that controls an elevator in a building. The scenarios of system are chosen in a way that can associate the complex behaviours of a sequence diagram and make use of the transformation rules defined in Chapter 5. The scenario of moving the elevator between floors and associated control functionalities are as follows. The present example modifies and extends a similar system as given in [Douglass, 2004, Fernandes et al., 2007, Radjenovic and Paige, 2010].

We assume that an elevator has an internal display panel with a set of numbered buttons each with a floor number and another three buttons for the functions open door, close door and alarm. This scenario is an attempt to replicate the elevator functions for one user. When a user enters the elevator, the first thing the elevator control does is to close the door. When at a floor, this can happen either by pressing the close button of the button panel or automatically by the elevator control after a timeout.

Once the door is closed, a user selects the floor that he/she wants to go to by pressing a numbered button on the button panel, and the corresponding floor number is passed on to the elevator system. The elevator starts to move until it reaches the corresponding requested floor. While the elevator is moving, pressing the open door button and associated interactions that cause the door to open are considered invalid executions and will not be executed. Further, the number of the passing floor is displayed on the button panel by the elevator.

When the elevator receives an arrival signal from a sensor (here an external instance) at the requested floor it stops, and the door *must* open representing a *mandatory* behaviour. Once the door opens it remains open for a fixed period of time. Yet again, the door is closed after the pre-defined time duration has expired or when the close door button is pressed by a user inside the elevator.

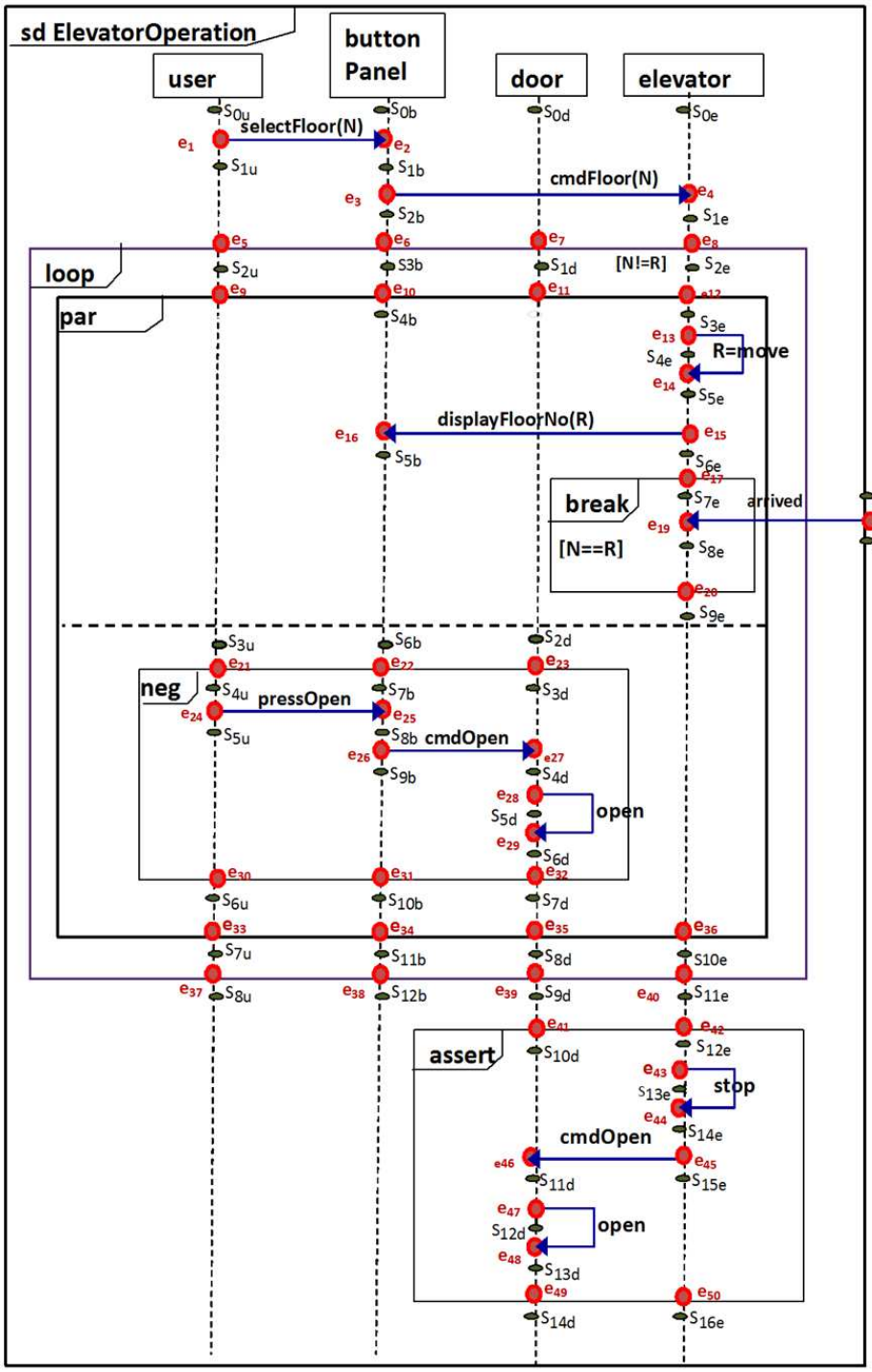


Figure 8.28: A SD for an example of an elevator system.

The interactions for the control functionalities of the elevator system are shown in the sequence diagram of Figure 8.28. Four instances are involved in the interaction shown: a **user**, a **buttonPanel**, a **door** and the **elevator** (control). The **buttonPanel** contains the floor numbers, the alarm, and the open/close buttons.

The diagram assumes that the elevator door is already closed and the user is inside the elevator. The interaction $SD_{ElevatorOperation}$ starts with the **user** pressing a floor numbered button on the **button panel** which is indicated by the local transition $selectFloor(N)$ where N indicates the selected floor number. This information is then forwarded to *elevator* by the local transition $cmdFloor(N)$. Here, we use the variable R associated with the instance *elevator*, to denote the current floor number while elevator is moving.

This is followed by a *loop* interaction fragment with the condition $[N! = R] == T$ (the requested floor is not equal to the current floor), indicating that the interactions within the fragment are repeated until the requested floor is reached; i.e. the current floor (value of R) is the same as the requested floor (value of N). The *loop* fragment starts with the self-transition *move* executed by the instance *elevator*.

A user is not allowed to open the door while the elevator is moving. Disallowed behaviour can be represented by a *neg* interaction fragment, where only the exact sequence of interactions contained in a *neg* fragment are disallowed. In our example, a user may press the open button as many times as he/she wants with no effect. Because the functions for pressing the button, forwarding the command to the door and opening the door is disallowed.

Further, while moving the elevator always displays the current floor number for each floor it passes. This is represented using the local transition $displayFloorNo(R)$ from *elevator* to *buttonPanel*. Since the disallowed be-

haviour and the floor display can happen in parallel, they are included in different operands in a *par* fragment.

The *loop* fragment includes a nested *break* fragment that includes the termination condition associated with *loop*. Here, the local transition with the message label *arrived* sent from a gate event to *elevator* indicates that the requested floor has been reached. Thus, when the break condition $[N == R]$ evaluates to true, the loop is exited and the remaining interactions in $SD_{ElevatorOperation}$ are carried out.

The *loop* fragment is followed by an *assert* fragment that indicates a mandatory behaviour. I.e., after the requested floor has been reached, the elevator must stop and open the door. The sequence of local transitions *stop*, *cmdOpen* and *open* placed inside an *assert* interaction fragment indicates that this sequence is compulsory and the only valid continuation. Overall, the diagram in Figure 8.28 contains several interaction fragments and is relatively complex. This diagram can be simplified by grouping sets of interactions into different diagrams and using *ref* interaction fragments to refer back to them.

The corresponding CPN model obtained by applying the defined transformation rules is shown in Figure 8.29. The model $CPN_{ElevatorOperation}$ contains colours, places and net transitions that correspond to the instances, state locations and local transitions of the diagram $SD_{ElevatorOperation}$, respectively.

The unlabelled net transitions indicate the synchronisation behaviour at the beginning and the end of each interaction fragment. The places and net transitions of the CPN in this example are represented using different shades (colours). This shading is only for the purpose of improving the readability of the CPN by highlighting and distinguishing the places of different object types (colours in the CPN terminology) and labelled/unlabelled net transitions. The colours or shades used in the figure have no semantic meaning.

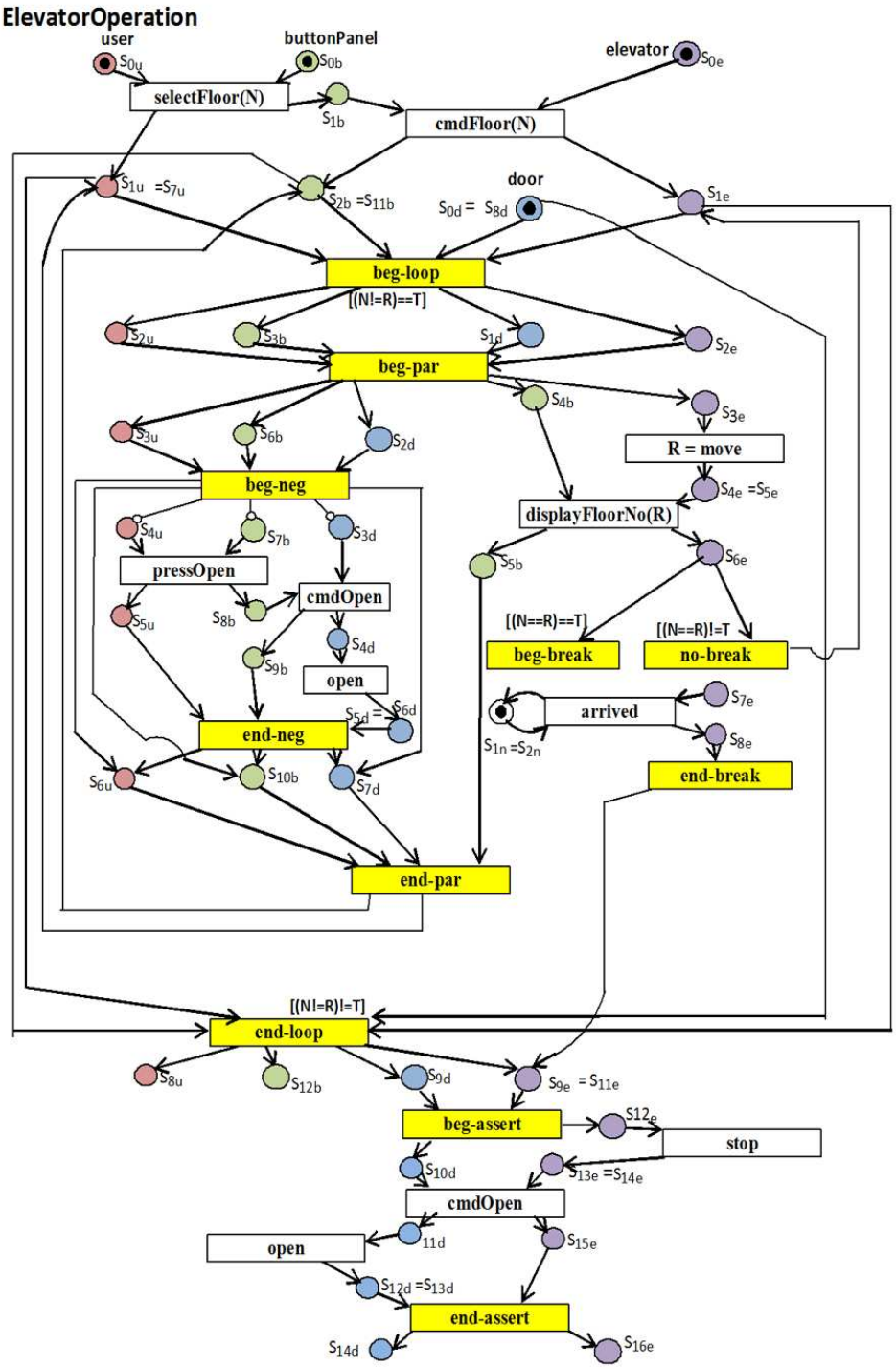


Figure 8.29: The corresponding CPN for the example of the elevator system.

In order to represent the iterative behaviour there is an equality between the places that correspond to the state location before the beginning of the *loop* fragment and the maximum state location within the operand for a given instance. This is shown by the place equalities $S_{1u} = S_{7u}$, $S_{2b} = S_{11b}$ and $S_{0d} = S_{8d}$. The net transitions $t_{beg-loop}$ and $t_{end-loop}$ are associated with the iterative condition ($N! = R$), that evaluates to true and false, respectively. Thus when the condition associated with the net transition $t_{beg-loop}$ evaluates to false, the control flow goes to the net transition $t_{end-loop}$.

Disallowed net transitions, in other words, net transitions that should not execute within the execution flow of the CPN, are represented within the net transitions $t_{beg-neg}$ and $t_{end-neg}$. Inhibitor arcs (represented by a line with a small circle at the end) are used to link $t_{beg-neg}$ and the places that correspond to the minimum state locations of the *neg* fragment. For example, the inhibitor arc for instance u, can be represented as: for $a \in A_{in}$, $node(a) = (t_{beg-neg}, S_{4u})$. This disables the token flow and the firing of transitions within the negative behaviour.

Instead, additional arcs are used to link $t_{beg-neg}$ with the places that correspond to the state locations after the *neg* fragment (i.e. the state locations that are given by the function θ). For example, formally for the colour u, $a_1 \in A$, and $node(a_1) = (t_{beg-neg}, S_{6u})$. Further, the status of the places within the negative behaviour is considered as unsafe and these places should not be reached in this CPN model.

The behaviour of the *break* fragment nested within the *par* fragment in the SD is modelled in the CPN by applying the transformation Rule 5.19. Since there are no other interactions after *break* and within the *loop*, an additional net transition $t_{no-break}$ is used in the CPN in order to maintain the control flow within the CPN, when the break condition is false. Thus $t_{no-break}$ is linked

to the place of the corresponding colour before the unlabelled net transition $t_{beg-loop}$. Here, the condition $(N == R)$ is used to check whether the elevator has reached the requested floor. The condition evaluating to true is associated with $t_{beg-break}$ and the negated disjunction of the expression is associated with $t_{no-break}$. Further, there is an equality between the place after $t_{end-break}$ and $t_{end-loop}$ such that $S_{9e} = S_{11e}$, in order to represent the termination of the iterative behaviour after the break.

The mandatory behaviour (i.e., when the elevator stops at the requested floor the door must open) is represented within the net transitions $t_{beg-assert}$ and $t_{end-assert}$. The status of the places within this assertion behaviour is *incomplete* and the next place with status *complete* is the target of the net transition $t_{end-assert}$. This indicates that all transitions within an assertion are required to fire, and if only a part of the net transitions have fired, the behaviour of the net remains incomplete. Conversely, we must ensure that all net transitions within an assert behaviour must be executed in order to complete the behaviour of the net.

Figure 8.30 shows a sequence diagram $SD_{closeDoor}$ that represents the interactions associated with a door closing functionality of an elevator. The SD contains four instances, namely *user*, *buttonPanel*, *door* and *elevator*. The two alternative ways for an elevator door to close are represented by the two operands of an *alt* interaction fragment. If the time is in between $[0, 3]$ time units, in the first case, the user presses the close button on the button panel given by the local transition *pressClose*, which triggers *sendClose* to be sent to *door*. Alternatively, after 3 time units of inactivity (timeout), the instance *elevator* sends a *cmdClose* message to *door*. In either way, when *door* receives a local transition *cmdClose*, it executes the local transition $t = (e_{15}, close, e_{16})$ that closes the door.

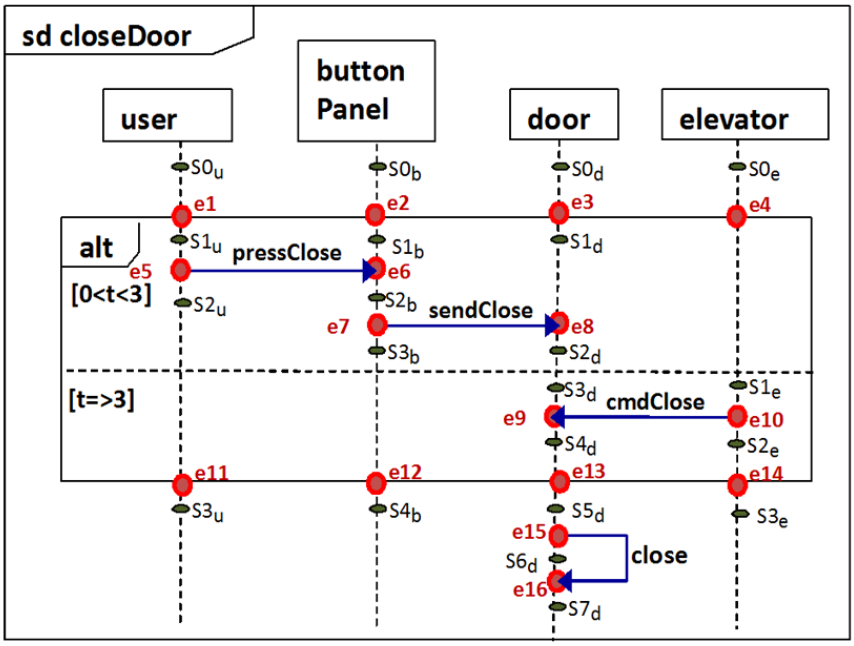


Figure 8.30: A SD for the door closing function of the elevator system.

The corresponding CPN representation for the interactions associated with $SD_{closeDoor}$ is shown in Figure 8.31 and generated using the transformation rules given in Chapter 5. By applying the general transformation rules, the CPN contain colours, namely *user*, *buttonPanel*, *door*, *elevator*, and the corresponding identifiers u, b, d, e, respectively. The beginning and the end of the *alt* interaction fragment are mapped to the net transitions $t_{beg-alt}$ and $t_{end-alt}$, respectively. The state locations and the local transitions of the SD are mapped onto the places and net transitions of the CPN respectively, and the arcs link places and net transitions as expected. Applying the transformation of an alternative behaviour (Rule 5.16), there is an equality between the places that correspond to the minimum state locations of the operands for a given instance. The same equality is applied for the maximum state locations. This is shown by the equalities $S1_d = S3_d$ and $S2_d = S4_d$.

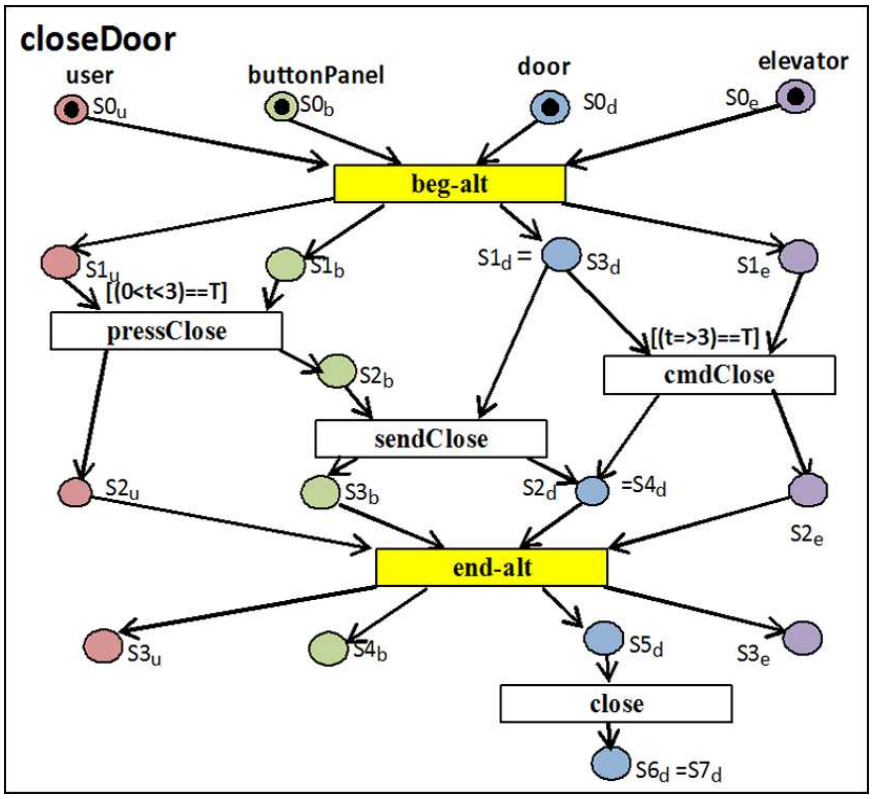


Figure 8.31: The corresponding CPN for the door closing function of the elevator system.

Further, the condition of each operand is associated with the net transition that corresponds to the first local transition with the operand. Formally, this can be represented as $guard(t_{pressClose}) = [(0 < t < 3) == T]$ and $guard(t_{cmdClose}) = [(t => 3) == T]$.

One of the main advantages of a CPN model is that a CPN model constitutes one single coherent description of the behaviour specified by the sequence diagram. The execution flow of a CPN shows how objects operate (communicate) with one another and their order of execution. CPN-based simulation facilitates the understanding and analysis of the system behaviour [Jensen and Kristensen, 2009] making it possible to detect obvious deadlocks, reachability

problems, and so on. It also shows clearly how the markings of places changes, how transitions become enabled, how tokens flow from a net transition to another, etc. For example, $CPNTools$ [Jensen et al., 2007, Jensen and Kristensen, 2009], which is a popular CPN software tool, makes it possible to model and check properties such as boundness (e.g., possible token colours, maximum number of tokens), liveness (whether transitions become enabled) and reachability states using *simulations* and *state space analysis* techniques [Jensen and Kristensen, 2009].

The identification of unreachable states (places in CPN) sometimes proves to be simply superfluous or otherwise signify an error in the design. Whatever the case, the occurrence of such situations requires design modifications. For example, in Figure 8.29 the places S_{3u},S_{4u},S_{4b},S_{5b},S_{6b},S_{2d},S_{3d}, S_{4d}, $S_{5d} \in P$ (that correspond to the state locations within *neg* fragment in Figure 8.28) should be unsafe and intentionally unreachable. This can be seen through simulation.

Additionally, CPN models can be used to identify deadlock situations and the design model can be subsequently modified to avoid such situations. Deadlock arises for example when two or more objects are waiting for each other to release a resource. Analysis methods such as simulation are by their nature operational. The control flow of the system behaviour can be illustrated by simulating the tokens as they are passed in the net from transition to transition. A token is passed only when a net transition can fire (is enabled). Thus, the CPN model can be shown to the user in order to reproduce the expected scenarios of the system behaviour and validate the original UML SD model. I.e. it makes it possible to find out what will happen when a system is executing. Here, we focus on simulation for the current example. However, notice that simulation is not exhaustive and may not discover all the problems

a model has, compared to formal verification by model checking [Nimiya et al., 2010, Yatake and Aoki, 2010, C.Baier and J.Katoen, 2008, Kwiatkowska et al., 2007, Grumberg and Long, 1991, Baier et al., 2007].

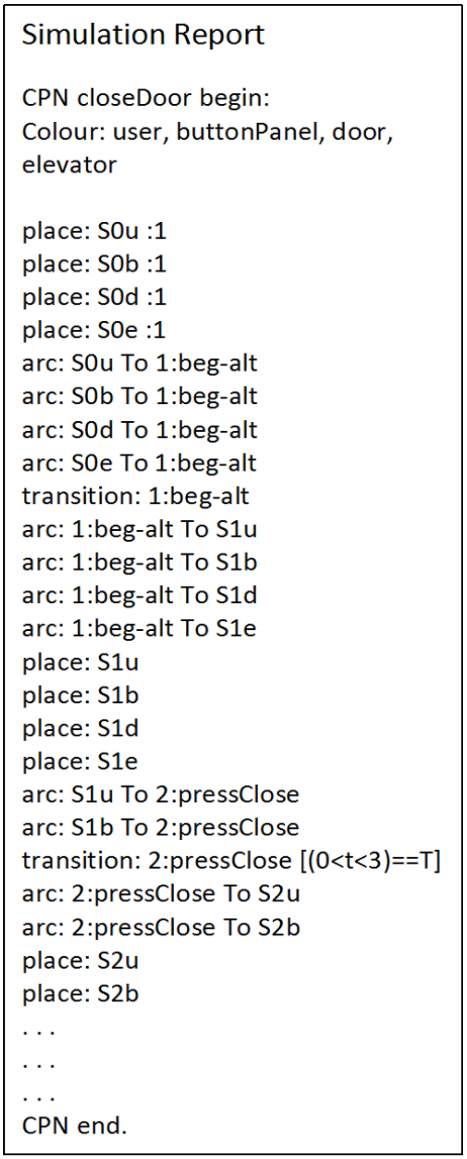
Here, we show a possible analysis of a SD by using our SD-to-CPN transformation to generate a CPN that can be simulated and analysed. Here, we show the execution of a CPN manually, but we note that this can be extended to an automated CPN analysis in the future. Figure 8.32(a) shows the simulation report for Figure 8.31 that was generated manually by considering the token flow and the textual grammar defined for CPNs in Section 8.4.2. This report shows the transitions that have occurred together with the places and arcs. Figure 8.32(b) shows the state space report for the CPN model in Figure 8.31 that has been manually generated by observing the behaviour of the CPN.

Since the simulation report is based on the token flow of the model and the corresponding textual grammar defined for the CPN, the analysis results comply with the available CPN simulation and analysis reports ([Jensen and Kristensen, 2009]). With this analysis it is possible to check properties of the design model. For example, reachability states can be extracted by referring to the place list in the simulation report. The liveness can be measured by considering the net transitions that are enabled. We can thus keep track of the actions that are executed and the states that are reached in a design model.

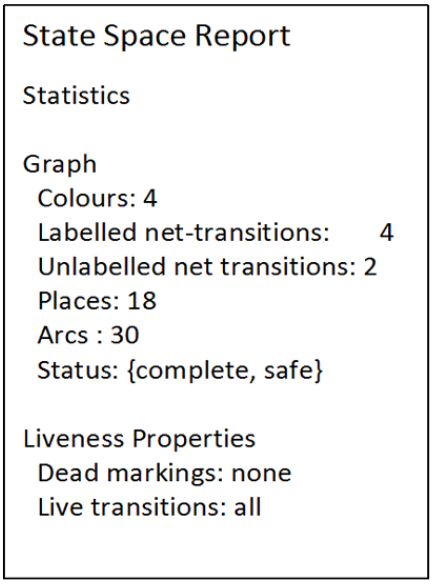
Since we consider the token flow of a CPN, it can be shown that the net transitions associated with an invalid behaviour are not included in the simulation report. For example, if we generate a simulation report for the CPN model shown in Figure 8.29, the net transitions *pressOpen*, *cmdOpen*, and *open* do not fire since the tokens do not pass to the specified negative behaviour. Here, after the net transition $beg - neg$ the next places are $S5_u$, $S7_b$ and $S6_d$.

These two reports can be used to locate errors or increase the confidence

in the correctness of the software system model. Further, these reports show the possibility of analysing the generated CPN models in a similar way to the existing CPN simulation and analysis tools.



(a)



(b)

Figure 8.32: The simulation report and the state space report generated from the CPN model.

8.7 Concluding Remarks

With the potential impact of model-driven approaches on software development practices, better tools are needed to automate the construction and evolution of software models. This chapter has shown that the implementation of our transformation rules from SDs to CPNs is possible and relatively straightforward. A complete implementation is beyond the scope of the present thesis.

Generally, for software tools to become truly useful in aiding developers, they need to be able to automate the models developed by them directly and also feed results back at the same level. In other words, users should be able to carry out the transformation and analysis of their original designs without expert knowledge of the formal models used underneath for the actually formal analysis. Therefore this prototype tool could be extended to incorporate such capabilities by back annotating the analysis results to the source SD model. Further, with the platform independent core implementations of the SD2CPN tool, it is possible to develop plugins for existing tools using the extensibility support of the tool given by the text-based grammar.

Additionally, this chapter has described the applicability of the defined transformation rules using two examples that covers different levels of system functionalities. Finally, for the elevator example we have showed the execution of a CPN manually, which can be extended to automated CPN analysis in future.

9 Discussion and Conclusion

The key contribution of this thesis is an approach to the formal representation of a behavioural parametric model to model transformation with a syntactical and semantical correctness proof. Two concrete popular models were chosen: a behavioural UML model and coloured Petri nets (CPNs). Both models have extensions for real-time and stochastic behaviours, which enhances their applicability. Our framework covers these extensions and extends our correctness proof accordingly. The benefits of a CPN target model are considerable with a rich and well-developed theory and a wide range of practical tools.

This chapter starts with a discussion of the motivation behind this thesis, gives an overview of each chapter and includes a careful evaluation of the outcomes and challenges for future work.

9.1 Discussion

Modern software systems in most domains are increasingly complex and need to function with high reliability. In particular, software applications with critical and real-time behaviours have high requirements on their dependability. The development of these complex software systems requires strong modelling and analysis methods including formal verification and quantitative modelling. Software development approaches following a Model Driven Development (MDD) perspective have widened the use of software models making models the core assets of the software development process and using model transformations to generate new models from the existing ones.

The object-oriented UML is a widely used intuitive but mostly an informal graphical modelling language for the design of complex systems. By contrast, formal models provide theoretical support that makes it possible to verify system designs. Consequently, formal models are essential to guarantee the cor-

rectness of systems and increase the trustworthiness of the developed systems with dependable requirements. Thus, it is immensely beneficial for modern complex software development needs to combine the benefits of formal and non-formal models. However, the transformation between those models has to be seamless with a well-established proof of correctness for the result of a formal analysis to propagate adequately to the informal design models.

The model transformations defined in this thesis have successfully bridged the gap between informal notation (UML SD and IOD) and formal notation (CPN and its variants) used for analysis purposes, with a complete proof of syntactic and semantic correctness of the transformation. Though the proof was done in the context of a SD to a CPN transformation, some of its underlying principles can be generalised to other behavioural transformations.

9.1.1 Research Summary

This research has been aimed at the development of a rigorous framework based on MDD that facilitates transformations of design models for verification. This thesis has focused on defining model transformation rules from a UML 2 sequence diagram (SD) to a CPN with the aim of enabling possible analyses over the CPNs and consequently enabling the formal verification of the UML design models.

Chapter 2 has described in considerable detail software design models and related formal model transformation approaches in order to identify a stable platform of knowledge that was explored and developed in this thesis. Here, the UML 2 sequence diagram, which is a commonly used diagram for capturing inter-object behaviour, is identified as the main non-formal design model for this thesis. CPNs, and its extensions for real-time, stochastic and hierarchical behaviours are selected as the underlying formal model, because they have a

rich and well-defined theory and tools, where in particular CPNs offer a natural support for object-oriented modelling by using colours to distinguish between object types.

Chapter 3 and 4 have described formal representations of SD and CPN models as the first step for applying formal processing with model transformations. These formal definitions incorporate general and complex behaviour of software systems and have been defined with extensibility and variability in mind. The formally defined syntax and semantics of SD and CPN models are loyal to the standardisations of the respective models. In order to model real-time, probabilities and complex structures, these formal representations were given a simple, but powerful, extension for the handing of time, stochastic, and hierarchical aspects, respectively. Here, we explored the flexible ways of adding these variants to a design and how that design can be added onto the target model with only minor adjustments. Moreover, we have defined the SDs with time, stochastic and hierarchical aspects, Interaction overview diagrams (IODs) and the corresponding formal models SCPN, TCPN and HCPN as the considered extensions of CPNs.

Additionally, we have defined languages (set of legal traces) of SDs and CPNs, in a way that the transformation rules can guarantee a direct correspondence between the set of legal traces of both models. That is, the languages are equivalent also known as strongly consistent. Consequently, we do not get implied behaviours in the synthesised CPN, which facilitates an accurate analysis on the given UML design models using any of the existing tools available for coloured Petri nets.

We are not aware of any other formal semantics for UML2 sequence diagrams and CPNs that supports the formal definition of model transformation rules, with the same strength and generality as of ours. Several other ap-

proaches exist, that consider the transformation from SD to CPNs, but all with some shortcomings. The most prominent being the lack of a formal proof of semantic correctness and a flexible parametric extension for real-time, stochastic and hierarchical behaviours. This is established in Chapter 7 of this thesis.

Exogenous M2M transformations are defined in Chapter 5 and 6, where CPN is the target model that can be used for a particular analysis approach. Chapter 5 has defined the transformation rules from a SD to a CPN for the mapping of general and complex behaviours such as alternative, parallel, iterative, forbidden, mandatory and critical. The flexible nature of the defined transformation allows us to extend the rules to different models and domains, conveniently.

In particular, Chapter 6 has defined partial and incremental transformation rules to transform a SD with reference behaviour to the corresponding HCPN. Additionally, model composition rules have been defined to obtain a single model from two or more related models for a unified understanding of the entire system. Moreover, given a SD with time and stochastic annotation, parametric transformations have been defined to obtain the corresponding TCPN and SCPN, respectively. Further, hierarchical transformation rules were defined between the pairs SDs, IOD and CPN, HCPN. These M2M transformations have been defined by creating a one-to-one correspondence between the elements in the source and target models.

In a model transformation framework, it is important that the transformation preserves the semantics of the source model. Without this any analysis result in the target model cannot be translated into the original source model in a meaningful manner. Chapter 7 has proved the syntactic and semantic correctness of the defined model transformation rules. The syntactic correct-

ness is shown using meta-model based graph grammar rules, which show that the transformation produces a well-formed target model from a valid source model. More importantly the semantic correctness is shown by proving a one-to-one correspondence between the legal traces of each model, and hence that the underlying languages of each model are strongly consistent or equivalent. This indicates not only that the behaviour of the source model is preserved in the target model, but also that the target model behaves exactly the same way as of the source model without additional or unexpected behaviours being possible in the target model. This result entails the preservation of other associated model transformation properties such as completeness, soundness, termination, and bisimulation.

The denotational model transformations defined the chapter 5 and 6 may not necessarily be appealing and human-friendly to a software developer that has experience in using IDE and CASE tool support. Chapter 8 has explained a prototype tool that implements the general transformation rules, in order to explore the possibility of developing a tool that supports automation of our model transformations. The backend of the tool is implemented considering the meta-models and the defined formal transformation rules. The motivation behind the tool has been to make the formal model transformations easily available and practical in use. For this reason this tool facilitates a graphical SD editor and a CPN representation that hides the underlying formal representation. Also, the tool has implemented textual representation for the considered models in order to integrate with other existing tools in future. However, this tool is a prototype only and developed as a proof of concept of our framework and is naturally far from the expected support for industrial norms.

Further, example-based case studies have been described to show the ap-

plicability of the defined transformations in practical use. Manual analysis of the synthesised CPN is used to illustrate the possible analysis over the formal model. Additionally, the manual analysis over the synthesised CPNs is used to illustrate the possible analysis over the formal model, for identifying the presence of model properties such as system flaws and reachability states, hence verifying the UML design models. This has increased the confidence of the correctness of the defined model transformations.

The work presented in this thesis is unique and new to the best of our knowledge in which the research findings show significant originality and contribution to the field of correct M2M transformations of behavioural software design models.

9.1.2 Research Contribution

This research successfully brought together different existing software design models such as UML 2 SDs and IODs, formal models such CPNs and its variants such as TCPN, SCPN, HCPN in order to enable different formal verification of system models in an underlying MDD approach. The main contributions are the following:

- The defined model transformation framework supports an MDD-based approach.

  The chosen non-formal and formal design models, SDs and CPNs, support for object-oriented software modelling and are capable of modelling the behaviour of systems at different levels of abstraction. The defined exogenous transformations can be reused with simple modifications when extending for different variants of source and target models. In addition, the defined framework supports modularity with composition and decomposition mechanisms. Further, this framework enables to analyse

the synthesised formal model with existing tools and verifies the system model before the actual implementation. Hence, the approach described in this thesis supports MDD.

- The considered models are capable of modelling complex systems with real-time and stochastic behaviour.

  The formal representation of SD defined in this thesis is capable of capturing additional behaviours such as critical, forbidden, stochastic and real-time. Moreover, we have defined parametric model transformations to synthesise formal models with time and stochastic behaviours. Further, the decomposition mechanisms give a powerful alternative to structuring interactions at different levels of abstraction and help to model large-scale, complex systems.

- The formal model transformation framework enables to improve the quality of the software system and avoid excessive costs.

  The transformation of non-formal design models into formal models enables formal analysis of the design models, hence increasing the trustworthiness of the developed systems by guaranteeing system correctness. By applying partial and incremental transformations to stepwise development allows partial analysis. Thus, early analysis of design models would identify possible flaws of the system and validate the design model. Consequently, this helps to eliminate excessive flaws, time and cost associated with the software development and enables quality improvements.

- The model transformation framework is flexible to use in practice, even for non-experts in formal methods.

  The flexible use of a model transformation approach depends on its appealing to a software developer that has experienced in IDE. The proto-

type tool has shown the possibility of automating the defined transformations with IDE support. The generated formal model can be used to analyse using the existing tools. Thus the users are able to carry out the transformations without expert knowledge on formal methods.

The overall statement of this thesis in Chapter 1 is that *the provision of the model transformation framework to enable different analysis and validate the model before the actual implementation, to develop software with less risk and cost*. We have shown the evidence in support of this statement with (1) theoretical underpinning by theories related to SD and CPN with variations and model transformations, (2) domain feasibility by performing system modeling on the selected design models, and (3) validity and acceptance with the mathematical proof strategies, case studies and prototype implementations of model transformations.

9.1.3 Research Challenges

The work carried out within this thesis had several challenges.

First, it was a challenge to choose a formal model that facilitates the analysis of a design model and can be used to formally verify the original design model. We have selected the CPN as the underlying formal model for this thesis as it is a well-known formal model with rich theory and practical applications that support for formal analysis of behavioural models. Further, CPNs offer a natural support for our approach when transforming object-oriented models, because the colours in CPNs can be used to distinguish between object types.

Moreover, some ambiguous and underspecified definitions given in the UML SD standards made it difficult to formalise the semantics for some interaction fragment behaviours such as *assert* and *neg*. This was addressed by consider-

ing the intended trace semantics of these behaviours.

When defining the languages associate with each model, it was a challenge to guarantee the preservation of the same behaviour in the target models as of the source model, which is essential for establishing the correctness of the model transformation process. We have shown the syntactic correctness by mapping an element between the considered meta-models. The semantic correctness is shown by proving a one-to-one correspondence between valid traces in the source and target models. Thus, we have shown that the languages are strongly consistent.

Further, it was a challenge to show the practical applicability of the transformation framework. We have shown the model transformations and manual analysis for the real world scenarios using example case studies. Additionally, the implemented prototype tool automates the transformation process and consists of a GUI that hides the formal details from the user, and the associated textual representations of models that enable integrations with existing tools.

9.1.4 Research Limitations

This research is one step towards establishing a MDD-based framework for the flexible analysis of complex interaction behaviour for software design models. Some assumptions and limitations are taken into account to narrow the scope of this thesis, fitting it into the available resources and constraints of the research.

We have mainly considered the formal representations and transformations for SD and CPN as well as some of the existing CPN extensions for real-time, stochastic and hierarchical behaviours. However, more formal model definitions and transformations can be defined to support a wide variety of

UML design models that represent different views of a system such as class diagrams and activity diagrams.

An automated CPN simulation and analysis that facilitates users with the reproduction of the expected scenarios and analysis of model properties is beyond the scope of this research. However, the case studies carried out in this thesis have demonstrated the manual analysis of traces over the synthesised CPN and the test runs generated by the tool have enabled the analysis of design properties such as reachability states, firing transitions and liveness.

The correctness proof has established language equivalence for source and target models. This guarantees the existence of a bidirectional transformation of our models. Even though, our transformation rules were defined in a SD-to-CPN way, the results of the formal analysis can be back-annotated to the UML models. Tool support for implementing this was however, not in the scope of the present work.

Finally, the prototype tool has implemented only the general transformation rules from a SD to a CPN and the quality of service requirements such as performance were not considered at this stage and can be developed further, as future work.

9.2 Further Work

The formal model transformation framework defined in this thesis brings formal models and techniques more naturally into MDD-based software development. This research and its outcomes could guide future researchers for possible further extensions, and MDD based software development as follows.

The defined MDD-based model transformation in this thesis (for UML SDs to CPNs with related extensions) is not sufficient for the modelling and analysis of software systems with different structural and behavioural views.

The framework can be extended with a family of transformations from non-formal UML models to different formal models, which can be analysed in various ways and ultimately validate the original design models.

This work can be extended with defining a repository of formal model definitions and transformations that support for a wide variety of design models with possible semantic variations that represents different views of a system [Cengarle et al., 2009]. For example, UML models can be transformed into PEPA nets that combine CPNs with the process algebra PEPA for modelling mobility, and analyse the performance of the designed system model [Gilmore et al., 2003a,Kloul and Kuster-Filipe, 2005,Bowles and Kloul, 2010,Tribastone and Gilmore, 2008]. Also, model integration rules can be defined between formal models, in order to analyse an entire software system with a complete set of models. This kind of model transformation and analysis framework would add flexibility to the framework and make it widely applicable for formal analysis of different models.

Moreover, the defined formal semantics can be used for several purposes such as a reference manual for the meaning of the model. Moreover, the formal representations can be used to check the consistency of the standardised semantics with the functionality offered by existing modelling tools. This would immensely beneficial for modern complex software development needs.

Another way of increasing the scope of this M2M transformation framework could be to investigate the applicability of parametric and incremental transformations that are proven to be correct, when there is a large variability in the target model, or when the semantic variability lies in the source model that links to research on language variability. The model transformation correctness proofs given in this thesis can be generalised in order to prove correctness and completeness of related families of transformations and models [Ehrig and

Ermel, 2008]. Establishing transformation correctness and completeness is important to ensure the results of formal analysis will not be invalidated by erroneous transformations as developers cannot distinguish whether an error is in the design or in the transformation. Also, correct model transformation enables to highlight the analysis results of the synthesised model back to the original design model. Back annotation of the analysis results can be performed using bidirectional transformations, since our result of strongly consistent languages already proves the existence of the bidirectional transformation rules.

Another consequence of this work is to perform formal analysis of the synthesised models that allows system verification. These techniques could facilitate to analyse model properties such as reachability of states, liveness, scalability, performance and to detect and avoid system flaws, hence, verify the correctness and completeness of the design models. The considered formal models with time and stochastic notion would be good candidates for implementing and performing analysis on such systems.

Additionally, with a family of transformations, we can explore and compare the results of different analysis and focus on the performance and scalability of existing tools [Kounev et al., 2010, Kounev and Buchmann, 2006, Jensen et al., 2007, Hinton et al., 2006].

From the tool point of view, the SD2CPN tool can be extended with more functionality that covers all the transformation rules and model analysis. The tool can be incorporated with extra functionalities such as syntax checking of the input UML model and feedback mechanism. Also, we could explore the practical use and quality of service requirements (usability, performance, scalability) of the tool. Further, the interoperability of this tool can be enhanced by implementing plug-ins that enable integration with existing tools.

Finally, the empirical support for the thesis statement can be strengthen

by performing model transformation of large real-world systems. Realistic case studies can be used to show the broad range of properties that can be analysed by transforming non-formal models into formal models. This will strengthen the empirical support for the considered model transformation and analysis framework.

9.3 Concluding Remarks

A significant amount of work has been conducted in this dissertation. A survey of the existing related work has provided a stable platform of knowledge that shaped the research towards achieving the objectives of this thesis.

This thesis has defined and described transformations from an informal graphical model into a formal model, and paved the way towards providing a correct model transformation framework. We have formalised UML 2 SDs, CPNs and some of its extensions, and defined formal transformation rules to obtain an equivalent CPN (or extension) from a given SD. We have proved that the defined languages of the models are strongly consistent, thus, the synthesised model is free of implied behaviours essential for an accurate analysis. The defined partial and incremental transformations contribute to a scalable approach for formal analysis. The parametric transformations help to transform models with real-time or stochastic behaviours. The transformations are seamless and transparent to software developers with no knowledge of the underlying model, and allow them to explore the benefits of the extensive suite of tools available for Petri nets.

The formal proofs give us the guarantee of the syntactic and semantic correctness of the transformations. The case studies were used to show the applicability of our approach and the transformation in real-world examples. The key capabilities of the SD2CPN tool have shown the possibility of imple-

menting the defined transformation using a MDD-based approach that allows easy extensions of rules to different models.

We believe that our MDD-based approach to behavioural model transformation is novel both in how to establish semantic correctness for transformations, and how to explore semantic variability in the target model for flexible formal analysis.

The thesis has achieved a framework for correct M2M transformations with semantic variability that enables possible formal analyses at the design level and thus validates the non-formal models. The transformation framework is easily extensible thus facilitating the support of specialised diagrams for different purposes including mobility, performance, real-time behaviour, and dependability.

References

[Abdallah et al., 2005] Abdallah, A., Bowen, J., and Nissanke, N. (2005). *Formal Methods for Safety Critical System, Chapter 9*. Wiley.

[Aburub et al., 2007] Aburub, F., Odeh, M., and Beeson, I. (2007). Modelling non-functional requirements of business processes. *Journal of Information and Software Technology*, 49(11-12):1162–1171.

[Akehurst et al., 2006] Akehurst, D., Bordbar, B., Evans, M., Howells, W., and McDonald-Maier, K. (2006). Sitra: Simple transformations in java. In *ACM/IEEE 9TH International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 351–364. Springer.

[Alhroob et al., 2010] Alhroob, A., Dahal, K., and Hossain, A. (2010). Transforming uml sequence diagram to high level petri net. In *Proceedings of the 2nd International Conference on Software Technology and Engineering(ICSTE)*, pages V1–260 – V1–264. IEEE Computer Society.

[Alur et al., 2003] Alur, R., Etessami, K., and Yannakakis, M. (2003). Inference of message sequence charts. In *Proceedings of the ICSE'00, IEEE Transactions on Software Engineering*, volume 29, pages 623–633. IEEE Computer Society.

[Ameedeen and Bordbar, 2008] Ameedeen, M. A. and Bordbar, B. (2008). A model driven approach to represent sequence diagrams as free choice petri nets. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 213–221.

[Ameedeen et al., 2009] Ameedeen, M. A., Bordbar, B., and Anane, R. (2009). A model driven approach to the analysis of timeliness properties. In *Pro-*

ceedings of the 5th edition of the European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), LNCS 5562, pages 221–236. Springer-Verlag.

[Ameedeen et al., 2011] Ameedeen, M. A., Bordbar, B., and Anane, R. (2011). Model interoperability via model driven development. *Journal of Computer and System Sciences*, 77(2):332–347.

[Amstel et al., 2007] Amstel, M. V., Lange, C., and Chaudron, M. (2007). Four automated approaches to analyze the quality of uml sequence diagrams. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 415–424.

[Anastasakis et al., 2010] Anastasakis, K., Bordbar, B., Georg, G., and Ray, I. (2010). On challenges of model transformation from uml to alloy. *Journal of Software and System Modeling*, 9(1):69–86.

[Anda et al., 2009] Anda, B., Hansen, K., and Sand, G. (2009). An investigation of use case quality in a large safety-critical software development project. *Journal of Information and Software Technology*, 51(12):1699–1711.

[Arlow and Neustadt, 2005] Arlow, J. and Neustadt, I. (2005). *UML 2 and the unified process : practical object-oriented analysis and design*. Addison-Wesley.

[Avizienis et al., 2004] Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33.

[Avizienis et al., 2001] Avizienis, A., Laprie, J. C., and Randell, B. (2001). Fundamental Concepts of Dependability. Research Report , LAAS-CNRS.

[Baier et al., 2007] Baier, C., Cloth, L., Haverkort, B. R., Kuntz, M., and Siegle, M. (2007). Model checking markov chains with actions and state labels. *IEEE Transactions on Software Engineering*, 33(4):209–224.

[Barbier and Cariou, 2008] Barbier, F. and Cariou, E. (2008). Component design based on model executability. In *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '08)*, pages 68–75.

[Baresi et al., 2011] Baresi, L., Morzenti, A., Motta, A., and Rossi, M. (2011). *From Interaction Overview Diagrams to Temporal Logic*, volume 6627 of *LNCS*. Springer-Verlag Berlin Heidelberg.

[Baresi and Pezz, 2005] Baresi, L. and Pezz, M. (2005). From graph transformation to software engineering and back. *In HJ Kreowski et al. (eds.) Formal Methods in Software and Systems Modeling*, pages 24–37.

[Benatallah et al., 2003] Benatallah, B., Chrzastowski-Wachtel, P., Hamadi, R., O'Dell, M., and Susanto, A. (2003). Hiword: A petri net-based hierarchical workflow designer. In *In Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD'03)*, pages 235–236. IEEE Computer Society Press.

[Benmerzoug et al., 2008] Benmerzoug, D., Kordon, F., and Boufada, M. (2008). A petri-net based formalisation of interaction protocols applied to business process integration. In *Proceedings of the 4th International Workshop on Enterprise and Organizational Modeling and Simulation (EOMAS 2008)*, pages 78–92. Springer-Verlag.

[Bernardi et al., 2002] Bernardi, S., Donatelli, S., and Merseguer, J. (2002). From uml sequence diagrams and statecharts to analysable petri net models.

In *Proceedings of the 3rd international workshop on Software and Performance*, pages 35–45.

[Beydeda et al., 2005] Beydeda, S., Book, M., and Gruhn, V. (2005). *Using Graph Transformation for Practical Model-Driven Software Engineering*. Springer Berlin Heidelbergh.

[Billington, 2004] Billington, J. (2004). ISO/IEC 15909-1:2004 Software and system engineering, High-level Petri nets, Part 1: Concepts, definitions and graphical notation. Technical Report .

[Billington and Reisig, 1996] Billington, J. and Reisig, W. (1996). Application and theory of petri nets 1996. In *Proceedings of the 17th International Conference*. LNCS.

[Bisztray et al., 2009] Bisztray, D., Heckel, R., and Ehrig, H. (2009). Compositionality of model transformations. *Electronic Notes in Theoretical Computer Science*, 236:5–19.

[Bobbio, 1990] Bobbio, A. (1990). System Modelling With Petri Nets. Technical Report , Istituto Elettrotecnico Nazionale Galileo Ferraris, Strada delle Cacce 91, 10135 Torino, Italy.

[Bondavalli et al., 2005] Bondavalli, A., Chiaradonna, S., and Giandomenico, F. (2005). *Model-Based Evaluation as a Support to the Design of Dependable Systems, Chapter 3*. Wiley.

[Boronat et al., 2005] Boronat, A., Carsí, J. A., and Ramos, I. (2005). An algebraic baseline for automatic transformations in mda. In *Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions (SETra 2004)*, volume 127, pages 31–47.

[Boronat et al., 2009a] Boronat, A., Heckel, R., and Meseguer, J. (2009a). Rewriting logic semantics and verification of model transformations. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE 2009)*, LNCS 5503, pages 18–33. Springer-Verlag.

[Boronat et al., 2009b] Boronat, A., Knapp, A., Meseguer, J., and Wirsing, M. (2009b). What is a multi-modeling language? In *Recent Trends in Algebraic Development Techniques*, pages 71–87. Springer.

[Bowles, 2006] Bowles, J. (2006). Modelling concurrent interactions. *Jounal of Theoretical Computer Science*, 351(2):203–220.

[Bowles and Bordbar, 2007] Bowles, J. and Bordbar, B. (2007). A formal model for integrating multiple views. In *Proceedings of the Seventh International Conference on Application of Concurrency to System Design (ACSD 2007)*, pages 71–79. IEEE Computer Society Press.

[Bowles and Kloul, 2010] Bowles, J. and Kloul, L. (2010). Synthesising pepa nets from iods for performance analysis. In *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering*. ACM.

[Bowles and Meedeniya, 2010] Bowles, J. and Meedeniya, D. (2010). Formal transformation from sequence diagrams to coloured petri nets. In *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC '10)*, pages 216 – 225. IEEE Computer Society.

[Bowles and Meedeniya, 2012a] Bowles, J. and Meedeniya, D. (2012a). Parametric transformations for flexible analysis. In *Proceedings of the 19th*

Asia Pacific Software Engineering Conference (APSEC '12), pages 634–643. IEEE Computer Society.

[Bowles and Meedeniya, 2012b] Bowles, J. K. F. and Meedeniya, D. (2012b). Strongly consistent transformation of partial scenarios. *SIGSOFT Software Engineering Notes (SEN)*, 37(4):1–8.

[Buyya et al., 2009] Buyya, R., Ranjan, R., and Calheiros, R. (2009). Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities. In *Proceedings of the 7th High Performance Computing and Simulation (HPCS 2009) Conference*.

[Cabot et al., 2010a] Cabot, J., Claris, R., Guerra, E., and de Lara, J. (2010a). Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302.

[Cabot et al., 2010b] Cabot, J., Clariso, R., Guerra, E., and de Lara, J. (2010b). A uml/ocl framework for the analysis of graph transformation rules. *Journal of Software and Systems Modeling*, 9(3):335–357.

[Cabot et al., 2010c] Cabot, J., Clariso, R., Guerra, E., and de Lara, J. (2010c). Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302.

[Cabot et al., 2008] Cabot, J., Clariso, R., and Riera, D. (2008). Verification of uml/ocl class diagrams using constraint programming. In *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW '08),*, pages 73–80.

[Campos and Merseguer, 2006] Campos, J. and Merseguer, J. (2006). On the integration of uml and petri nets in software development. In *Proceedings*

of the 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2006), LNCS 4024, pages 19–36.

[Carnevali et al., 2008] Carnevali, L., Grassi, L., and Vicario, E. (2008). A tailored v-model exploiting the theory of preemptive time petri nets. In *Proceedings of the Ada Europe: 13th Intternational Conference on Relieable SW Technologies*, volume 5026 of *LNCS*, pages 87–100. Springer-Verlag.

[Carnevali et al., 2009] Carnevali, L., Grassi, L., and Vicario, E. (2009). State-density functions over dbm domains in the analysis of non-markovian models. *IEEE Transaction on Software Engineering*, 35(2):178–194.

[Cavarra and Filipe, 2004] Cavarra, A. and Filipe, J. (2004). Combining sequence diagrams and ocl for liveness. In *Proceedings of the Semantic Foundations of Engineering Design Languages (SFEDL)*.

[Cavarra and Küster-Filipe, 2004] Cavarra, A. and Küster-Filipe, J. (2004). Formalizing liveness enriched sequence diagrams using ASMs. In *Proceedings of the 11th International Workshop on Abstract State Machines 2004 (ASM 2004) & in W. Zimmermann and B. Thalheim (eds.) Abstract State Machines*, LNCS, pages 62–77. Springer.

[C.Baier and J.Katoen, 2008] C.Baier and J.Katoen (2008). *Principles of Model Checking*. The MIT Press.

[Cengarle et al., 2006] Cengarle, M., Graubmann, P., and S.Wagner (2006). Semantics of uml 2.0 interactions with variabilities. In *Electronic Notes in Theoretical Computer Science*, 160, page 141155.

[Cengarle et al., 2009] Cengarle, M., Gronniger, H., and Rumpe, B. (2009). Variability within modeling language definitions. In *Proceedings of the MODELS 2009*, LNCS 5795, pages 670–684. Springer-Verlag.

[Cengarle and Knapp, 2004] Cengarle, M. and Knapp, A. (2004). Uml 2.0 interactions: semantics and refinement. In *Proceedings of the 3rd International Workshop on Critical Systems Development with UML (CSDUML04)*, page 8599.

[Christensen, 2002] Christensen, S. (2002). *Coloured Petri Nets Theory and Applications: Modelling and Verifications of Protocols.* available from www.mnlab.cs.depaul.edu/seminar/fall2002/CPN/pdf.

[Christensen and Petrucci, 2000] Christensen, S. and Petrucci, L. (2000). Modular analysis of petri nets. *The Computer Journal*, 43(3):224–242.

[Chrzastowski-Wachtel et al., 2003] Chrzastowski-Wachtel, P., Benatallah, B., Hamadi, R., O'Dell, M., and Susanto, A. (2003). A top-down petri net-based approach for dynamic workflow modeling. In *Proceedings of the International Conference on Business Process Management (BPM'03)*, LNCS 2678, pages 336–353. Springer Verlag.

[Cimatti et al., 2011] Cimatti, A., Roveri, M., Susi, A., and Tonetta, S. (2011). Formalizing requirements with object models and temporal constraints. *Journal of Software and Systems Modeling*, 10(2):147–160.

[Cohen et al., 1986] Cohen, B., Harwood, W., and Jackson, M. (1986). *The specification of complex systems.* Addison-Wesley.

[Cuadrado et al., 2011] Cuadrado, J. S., Guerra, E., and de Lara, J. (2011). Generic model transformations: Write once, reuse everywhere. In *Pro-*

ceedings of the International Conference on Model Transformation (ICMT 2011), LNCS 6707, pages 62–77. Springer-Verlag.

[D. Harel, 2003] D. Harel, R. M. (2003). *Come, Let's Play: Scenario-Based Programming using LSCs and the the Play Engine*. Springer.

[Dan et al., 2007] Dan, H., Hierons, R., and Counsell, S. (2007). A thread-tag based semantics for sequence diagrams. In *Proceedings of the Software Engineering and Formal Methods (SEFM 2007)*, page 173182. IEEE Computer Society.

[Dang et al., 2010] Dang, D., Truong, A., and M.Gogolla (2010). Checking the conformance between models based on scenario synchronization. *Journal of Universal Computer Science*, 16(17):2293–2312.

[de Alfaro et al., 1998] de Alfaro, L., Manna, Z., and Sipma, H. (1998). Decomposing, transforming and composing diagrams: The joys of modular verification. Technical Report.

[de Lara and Guerra, 2005] de Lara, J. and Guerra, E. (2005). Formal support for model driven development with graph transformation techniques. In *Proceedings of the Workshop on Development of Model Driven Software, MDA and Applications (DSDM'05)*, volume 157.

[Delatour and Lamotte, 2003] Delatour, J. and Lamotte, F. (2003). Argopn: a case tool merging uml and petri nets. In *Proceedings of the 1st International Workshop on Validation and Verification of software for Enterprise Information Systems: VVEIS 2003*, pages 94–102.

[Derrick and Wehrheim, 2010] Derrick, J. and Wehrheim, H. (2010). Model transformations across views. *Journal of Science of Computer Programming*, 75(3):192–210.

[Dinh-Trong et al., 2006] Dinh-Trong, T., Ghosh, S., and France, R. B. (2006). A systematic approach to generate inputs to test uml design models. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE '06)*, pages 95 – 104.

[dos S. Soares and Vrancken, 2008] dos S. Soares, M. and Vrancken, J. (2008). A meta modeling approach to transform uml 2.0 sequence diagrams to petri nets. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 159–164. ACTA Press Anaheim.

[Douglass, 2004] Douglass, B. (2004). *Real Time UML Third Edition: Advances in the UML for Real-Time Systems*. Addison-Wesley, 3 edition.

[Ehrig et al., 2008] Ehrig, H., Ehrig, K., and Hermann, F. (2008). From model transformation to model integration based on the algebraic approach to triple graph grammars. In *Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, volume 10.

[Ehrig and Ermel, 2008] Ehrig, H. and Ermel, C. (2008). Semantical correctness and completeness of model transformations using graph and rule transformation. In *Proceedings of the 4th International Conference on Graph Transformation (ICGT 2008)*, LNCS 5214, pages 194–210. Springer-Verlag.

[Eichner et al., 2005] Eichner, C., Fleischhack, H., Meyer, R., Schrimpf, U., and Stehno, C. (2005). Compositional semantics for uml 2.0 sequence diagrams using petri nets. In *Proceedings of the 12th International SDL Forum: Model Driven Systems Design*, volume 3530 of *LNCS*, pages 133–148.

[Elkoutbi and Keller, 1998] Elkoutbi, M. and Keller, R. K. (1998). Modeling interactive systems with hierarchical colored petri nets. In *Proceedings of the Advanced Simulation Technologies Conference*, pages 432–437.

[Emadi, 2010] Emadi, S. (2010). Mapping annotated sequence diagram to a petri net notation for reliability evaluation. In *Proceedings of the 2nd international Conference on Education Technology and Computer (ICETC)*, volume 3, pages 57–61.

[Emadi and Shams, 2009a] Emadi, S. and Shams, F. (2009a). Mapping annotated use case and sequence diagram to a petri net notation for performance evaluation. In *Proceedings of the 2nd international Conference on Computer and Electrical Engineering (ICCEE)*, pages 68–71. IEEE Computer Society.

[Emadi and Shams, 2009b] Emadi, S. and Shams, F. (2009b). Transformation of use case and sequence diagrams to petri nets. In *Proceedings of the ISECS International Colloquium on Computing, Communication, Control, and Management (ICCCM 2009)*, volume 4, pages 399–403.

[Fehling, 1993] Fehling, R. (1993). *A Concept of Hierarchical Petri Nets with Building Blocks: Applications and Theory of Petri Nets 1991*, volume 674 of *LNCS*. Springer.

[Fernandes et al., 2007] Fernandes, J., Tjell, S., Jorgensen, J., and Ribeiro, O. (2007). Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In *Proceedings of the Sixth International Workshop on Scenarios and State Machines (SCESM '07)*. IEEE Computer Society.

[Fernández et al., 2011] Fernández, E. B., Washizaki, H., Yoshioka, N., and VanHilst, M. (2011). An approach to model-based development of secure and reliable systems. In *Proceedings of the 6th International Conference on Availability, Reliability and Security (ARES 2011)*, pages 260–265. IEEE Computer Society.

[Firley et al., 1999] Firley, T., Huhn, M., Diethers, K., Gehrke, T., and Goltz, U. (1999). Timed sequence diagrams and tool-based analysis – a case study. In *The Second International Conference on The Unified Modeling Language, Beyond the Standard (UML'99)*, volume 1723 of *LNCS*, pages 645–660. Springer.

[France and Rumpe, 2007] France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Proceedings of the Future of Software Engineering, FOSE '07*, pages 37–54.

[Garousi, 2010] Garousi, V. (2010). Experience and challenges with uml-driven performance engineering of a distributed real-time system. *Journal of Information and Software Technology*, 52(6):625–640.

[Genrich and Lautenbach, 1981] Genrich, H. and Lautenbach, K. (1981). System modelling with high level petri nets. *Theoretical Computer Science*, 13:109–136.

[Gherbi and Khendek, 2006] Gherbi, A. and Khendek, F. (2006). Uml profiles for real-time systems and their applications. *Journal of Object Technology*, 5(4):149–169.

[Gilmore et al., 2003a] Gilmore, S., Hillston, J., Kloul, L., and Ribaudo, M. (2003a). Pepa nets: A structured performance modelling formalism. *Journal of Performance Evaluation*, 54(2):79–104.

[Gilmore et al., 2003b] Gilmore, S., Hillston, J., Kloul, L., and Ribaudo, M. (2003b). Pepa nets: a structured performance modelling formalism. *Journal of Performance Evaluation*, 54(2):79–104.

[Goknil et al., 2011] Goknil, A., Kurtev, I., van den Berg, K., and Veldhuis, J. (2011). Semantics of trace relations in requirements models for consis-

tency checking and inferencing. *Journal of Software and Systems Modeling*, 10(1):31–54.

[Gorton, 2006] Gorton, I. (2006). *Essential software architecture*. Springer.

[Greenyer and Kindlev, 2007] Greenyer, J. and Kindlev, E. (2007). Reconciling tgg with qvt. *G. Engels et al. (Eds.): MoDELS 2007*, pages 16–30.

[Grónmo and Móller-Pedersen, 2010] Grónmo, R. and Móller-Pedersen, B. (2010). From uml 2 sequence diagrams to state machines by graph transformation. *Proceedings of the Third international conference on Theory and practice of model transformations (ICMT'10)*, pages 93–107.

[Gronniger and Rumpe, 2011] Gronniger, H. and Rumpe, B. (2011). Modeling language variability. In *Proceedings of the Monterey Workshop 2010*, LNCS 6662, pages 17–32. Springer-Verlag.

[Grosu and Smolka, 2005] Grosu, R. and Smolka, S. (2005). Safety-liveness semantics for uml 2.0 sequence diagrams. In *Proceedings of the 5th international conference on application of concurrency to system design (ACSD'05)*, pages 6–14. IEEE Computer Society.

[Grumberg and Long, 1991] Grumberg, O. and Long, D. E. (1991). Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16.

[Haas, 2002] Haas, P. (2002). *Stochastic Petri nets: modelling, stability, simulation*. Springer.

[Halvorsen et al., 2007] Halvorsen, O., Runde, R. K., and Haugen, Ø. (2007). Time exceptions in sequence diagrams. *T. Khne (Ed.), MoDELS 2006 Workshops*, 4364:131–142.

[Hamadi and Benatallah, 2003] Hamadi, R. and Benatallah, B. (2003). A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference, (ADC'03), CRPIT 17*, pages 191–200.

[Hammal, 2006] Hammal, Y. (2006). Branching time semantics for uml 2.0 sequence diagrams. In *Proceedings of the Formal Techniques for Networked and Distributed Systems & E. Najm et al. (Eds.) FORTE 2006*, LNCS 4229, page 259274. Springer.

[Harel and Kugler, 2002] Harel, D. and Kugler, H. (2002). Synthesizing state-based object systems from lsc specifications. *Journal of Foundations of Computer Science*, 13(1):5–51.

[Harel et al., 2005] Harel, D., Kugler, H., and Pnueli, A. (2005). Synthesis revisited: Generating statechart models from scenario-based requirements. In *In H.-J. Kreowski et al, (eds.) Formal Methods in Software and System Modeling*, LNCS 3393, pages 309–324. Springer-Verlag.

[Harel and Maoz, 2007] Harel, D. and Maoz, S. (2007). Assert and negate revisited: Modal semantics for uml sequence diagrams. *Jounal of Software and System Model*, 7(2):237–253.

[Haugen et al., 2005] Haugen, Ø., Husa, K., Runde, R., and Stølen, K. (2005). Stairs towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 4(4):1–13.

[Haugen et al., 2006] Haugen, Ø., Husa, K., Runde, R., and Stølen, K. (2006). Why timed sequence diagrams require three-event semantics. Technical Report, University of Oslo.

[Heiner et al., 2007] Heiner, M., Ricter, R., Schwarick, M., and Rohr, C. (2007). Snoopy - a tool to design and execute graph-based formalisms. In *Proceedings of the AWPN Workshop*.

[Hermann et al., 2010] Hermann, F., Ehrig, H., Orejas, F., and Golas, U. (2010). Formal analysis of functional behaviour for model transformations based on triple graph grammars. In *Proceedings of the 5th International Conference on Graph Transformations*, LNCS, pages 155–170. Springer-Verlag.

[Hidaka et al., 2009] Hidaka, S., Hu, Z., and H. Kato, K. N. (2009). A compositional approach to bidirectional model transformation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 235–238. IEEE Xplore.

[Hillston and Kloul, 2006] Hillston, J. and Kloul, L. (2006). *A function-equivalent components based simplification technique for PEPA models*, volume 4054 of *LNCS*. Springer-Verlag.

[Hinton et al., 2006] Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D. (2006). Prism: A tool for automatic verification of probabilistic systems. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3920, pages 441–444.

[Holzmann, 1997] Holzmann, G. J. (1997). The model checker spin. In *Proceedings of the IEEE Transactions on Software Engineering*, volume 23, pages 279–295.

[Hülsbusch et al., 2010a] Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., and Wehrheim, H. (2010a). Full semantics preservation in model transformation- a comparison of proof techniques. Technical report

tr-ctit-10-09, Centre for Telematics and Information Technology, University of Twente.

[Hülsbusch et al., 2010b] Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., and Wehrheim, H. (2010b). Showing full semantics preservation in model transformation - a comparison of techniques. In *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM 2010)*, volume 6396 of *LNCS*, pages 183–198. Springer Verlag.

[Idani and Ledru, 2006] Idani, A. and Ledru, Y. (2006). Dynamic graphical uml views from formal b specifications. *Journal of Information and Software Technology*, 48(3):154–169.

[ITU, 1999] ITU (1999). Recommendation Z.120: Message Sequence Chart (MSC). Technical Report.

[Jensen, 1981] Jensen, K. (1981). Coloured petri nets and the invariant method. *Journal of Theoretical Computer Science*, 14:317–336.

[Jensen, 1990] Jensen, K. (1990). Coloured petri nets: A high-level language for system design and analysis. volume 483 of *LNCS*. Springer Verlag.

[Jensen, 1994] Jensen, K. (1994). An introduction to the theoretical aspects of coloured petri-nets. In *In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): ADecade of Concurrency*, volume 803 of *LNCS*, pages 230–272. Springer-Verlag.

[Jensen, 1997a] Jensen, K. (1997a). A brief introduction to coloured Petri nets. In *Proceeding of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97) Workshop*, volume 1217 of *LNCS*, pages 203–208. Springer-Verlag.

[Jensen, 1997b] Jensen, K. (1997b). *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer-Verlag.

[Jensen, 1998] Jensen, K. (1998). An introduction to the practical use of coloured petri-nets. In *In: W. Reisig and G. Rozenberg (eds.): Lectures on Petri Nets II: Applications*, volume 1492 of *LNCS*, pages 237–292. Springer-Verlag.

[Jensen et al., 2007] Jensen, K., Kristensen, L., and Wells, L. (2007). Coloured petri nets and cpn tools for modelling and validation of concurrent systems. In *International journal of Software tools technology transfer (STTT)*, volume 9, pages 213–254. Springer.

[Jensen and Kristensen, 2009] Jensen, K. and Kristensen, L. M. (2009). *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer-Verlag.

[Jones, 1990] Jones, C. (1990). *Systematic Software Development using VDM*. Prentice Hall.

[Karsai and Narayanan, 2008] Karsai, G. and Narayanan, A. (2008). Towards verification of model transformations via goal-directed certification. In Broy, M., Krger, I., and Meisinger, M., editors, *Model-Driven Development of Reliable Automotive Services*, volume 4922 of *LNCS*, pages 67–83. Springer Berlin Heidelberg.

[Katoen, 2008] Katoen, J. (2008). Perspectives in probabilistic verification. In *Proceedings of the 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '08)*, pages 3–10.

[Kavimandan and Gokhale, 2007] Kavimandan, A. and Gokhale, A. (2007). A parameterized model transformations approach for automating middleware

qos configurations in distributed real-time and embedded systems. In *Proceedings of the workshop on Automating service quality: International Conference on Automated Software Engineering (ASE'07)*, pages 16–21. ACM.

[Kavimandan and Gray, 2011] Kavimandan, A. and Gray, J. (2011). Managing the quality of software product line architectures through reusable model transformations. In *Proceedings of the 7th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. ACM publication.

[Kerkouche et al., 2010] Kerkouche, E., Chaoui, A., Bourennane, E., and Labbani, O. (2010). A uml and colored petri nets integrated modeling and analysis : Approach using graph transformation. volume 9, pages 25–43. ETH Zurich.

[Kessentini et al., 2010a] Kessentini, M., Bouchoucha, A., Sahraoui, H., and Boukadoum, M. (2010a). Example-based sequence diagrams to colored petri nets transformation using heuristic search. In *Proceedings of the 6th European conference on Modelling Foundations and Applications*, ECMFA'10, pages 156–172. Springer-Verlag.

[Kessentini et al., 2010b] Kessentini, M., Sahraoui, H., and Boukadoum, M. (2010b). Sequence diagram to colored petri nets transformation testing: an immune system metaphor. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON '10)*, pages 72–85. ACM.

[Khadka and B.Mikolajczak, 2007] Khadka, B. and B.Mikolajczak (2007). Transformation from live sequence charts to colored petri nets. In *Proceedings of the 2007 Summer Computer Simulation Conference*, pages 673–680. Society for Computer Simulation International.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Object Technology.

[Kloul and Kuster-Filipe, 2005] Kloul, L. and Kuster-Filipe, J. (2005). From interaction overview diagrams to pepa nets. In *Proceedings of the 4th Workshop on Process Algebras and Timed Activities (PASTA'05)*.

[Kloul and Kuster-Filipe, 2006] Kloul, L. and Kuster-Filipe, J. (2006). Modelling mobility with uml2.0 and pepa nets. In *Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD 2006)*, pages 153–162. IEEE Computer Society.

[Koch, 2006] Koch, N. (2006). Transformation techniques in the model-driven development process of uwe. In *Proceedings of the 2nd Model-Driven Web Engineering 2006 Workshop (MDWE'06)*. ACM DL.

[Kong et al., 2009] Kong, J., Zhang, K., Dong, J., and Xu, D. (2009). Specifying behavioral semantics of uml diagrams through graph transformations. *Journal of Systems and Software*, 82(2):292–306.

[Konigs, 2005] Konigs, A. (2005). Model transformation with triple graph grammars.

[Kounev and Buchmann, 2006] Kounev, S. and Buchmann, A. (2006). Simqpn–a tool and methodology for analyzing queueing petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364 – 394.

[Kounev and Dutz, 2007] Kounev, S. and Dutz, C. (2007). Queuing Petri Net Model Enviornment (QPME 1.0). User Guide : A Software tool for performance modeling and analysis using Queuing Petri nets .

[Kounev et al., 2006] Kounev, S., Dutz, C., and Buchmann, A. (2006). Qpme: Queueing petri net modeling enviornment. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QEST '06)*. IEEE.

[Kounev et al., 2010] Kounev, S., Spinner, S., and Meier, P. (2010). Qpme 2.0 - a tool for stochastic modeling and analysis using queueing petri nets. In *Proceedings of the Informations technologien fr die Praxis'2010*, pages 293–311.

[Kristensen et al., 1998] Kristensen, L., Christensen, S., and Jensen, K. (1998). The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2:98–132.

[Kristensen et al., 2004] Kristensen, L., Jørgensen, J., and Jensen, K. (2004). Application of coloured petri nets in system development. In *Proceedings of 4th Advanced Course on Petri Nets*, volume 3098, pages 626–685. LNCS, Springer-Verlag.

[Kuster et al., 2004] Kuster, J. M., Sendall, S., and Wahler, M. (2004). Comparing two model transformation approaches. In *Proceedings of the UML 2004 Workshop OCL and Model Driven Engineering*.

[Kuznetsov, 2007] Kuznetsov, M. B. (2007). Uml model transformation and its application to mda technology. *Journal of Programming and Computer Software*, 33(1):44–53.

[Kwiatkowska et al., 2007] Kwiatkowska, M., Norman, G., and Parker, D. (2007). Stochastic model checking. In *Proceedings of the Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, LNCS 4486, pages 220–270. Springer.

[Lakos and Petrucci, 2004] Lakos, C. and Petrucci, L. (2004). Modular analysis of systems composed of semiautonomous subsystems. In *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD'2004)*, pages 185–194. IEEE Computer Society Press.

[Laleau and Polack, 2008] Laleau, R. and Polack, F. (2008). Using formal metamodels to check consistency of functional views in information systems specification. *Journal of Information and Software Technology*, 50(7-8):797–814.

[Lano, 2009] Lano, K. (2009). *UML 2 Semantics and Applications*. Wiley.

[Le et al., 2010] Le, T., Palopoli, L., Passerone, R., Ramadian, Y., and Cimatti, A. (2010). Parametric analysis of distributed firm real-time systems: A case study. In *Proceedings of the ETFA'10*. IEEE Computer Society Press.

[Li et al., 2004] Li, X., Liu, Z., and He, J. (2004). A formal semantics of uml sequence diagram. In *Proceedings of the 15th Australian Software Engineering Conference*, pages 168–177. IEEE Computer Society.

[Limaa et al., 2009] Limaa, V., Talhia, C., Mouheba, D., Debbabia, M., Wanga, L., and M.Pourzandib (2009). Formal verification and validation of uml 2.0 sequence diagrams using source and destination of messages. In *Proceedings of the 4th International Workshop on Systems Software Verification (SSV 2009)*, volume 254 of *Electronic Notes in Theoretical Computer Science*, pages 143–160.

[Ludewig, 2003] Ludewig, J. (2003). Models in software engineering—an introduction. *Journal on Software and Systems Modeling*, 2(1):5–14.

[Lund, 2008] Lund, M. (2008). *Operational analysis of sequence diagram specifications*. PhD thesis, University of Oslo.

[Lund and Stølen, 2006] Lund, M. and Stølen, K. (2006). Deriving tests from uml 2.0 sequence diagrams with neg and assert. In *Proceedings of the 2006 international Workshop on Automation of Software Test*, pages 22–28.

[M. Nielsen, 1980] M. Nielsen, G. Plotkin, G. W. (1980). Petri nets, event structures and domains. In *Part I. TCS*, volume 13, pages 85–108.

[Mallet et al., 2006] Mallet, F., Peraldi-Frati, M., and André, C. (2006). From uml to petri nets for non functional property verification. In *Proceedings of the IEEE International Symposium on Industrial Embedded Systems*, pages 1–9.

[Mellor et al., 2004] Mellor, S., K.Scott, Uhl, A., and D.Weise (2004). *MDA Distelled; Principles of Model-Driven Architecture*. Addison Wesley.

[Mens et al., 2005] Mens, T., Grop, P. V., and Karsai, G. (2005). Applying a model transformation taxonomy to graph transformation technology. In *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 143–159. Elsevier publication.

[Merseguer and Campos, 2004] Merseguer, J. and Campos, J. (2004). Software performance modelling using uml and petri nets. In *Proceedings of the MASCOTS Tutorials 2003*, LNCS 2965, pages 265–289.

[Meyer, 2006] Meyer, B. (2006). Dependable software. *Journal of Dependable Systems: Software, Computing, Networks*, 4028:1–33.

[Microsoft,] Microsoft. Windows azure cloud platform online.

[Micskei and Waeselynck, 2010] Micskei, Z. and Waeselynck, H. (2010). The many meanings of uml 2 sequence diagrams: a survey. In *Software System Model*. Springer.

[Milner, 2009] Milner, R. (2009). *The Space and Motion of Communicating Agents*.

[Mosbahi et al., 2011] Mosbahi, O., Ayed, L. J. B., and Khalgui, M. (2011). A formal approach for the development of reactive systems. *Journal of Information and Software Technology*, 53(1):14–33.

[Moschoyiannis et al., 2009] Moschoyiannis, S., Krause, P., and Shields, M. W. (2009). A true-concurrent interpretation of behavioural scenarios. *In Proceedings of the 4th International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA@ETAPS '07)*, 203(7):3–22.

[Moschoyiannis et al., 2010] Moschoyiannis, S., Razavi, A., and Krause, P. (2010). Transaction scripts: Making implicit scenarios explicit. *In Proceedings of 5th International Workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA@ETAPS'08)*, 238:63–79.

[Moschoyiannis et al., 2005] Moschoyiannis, S., Shields, M. W., and Krause, P. J. (2005). Modelling component behaviour with concurrent automata. *In Proceedings of ETAPS 2005 workshop on Formal Foundations of Embedded Software and Component-based Software Architectures (FESCA'05)*, 141(3):199–220.

[MSDN-library,] MSDN-library. What is a model. Technical Report , http://msdn.microsoft.com/en-us/library/dd129503(VS.85).aspx.

[Murata, 1989] Murata, T. (1989). Petri nets: properties, analysis and application. In *Proceedings of IEEE Journal*, volume 77, pages 541–580. s.

[Naumenko and Wegmann, 2002] Naumenko, A. and Wegmann, A. (2002). A metamodel for the unified modeling language. In *Proceedings of the 5th International Conference on the Unified Modeling Language*, volume 2460 of *LNCS*, pages 2–17. Springer-Verlag.

[Nimiya et al., 2010] Nimiya, A., T.Yokogawa, Miyazaki, H., S.Amasaki, Y.Sato, and M.Hayase (2010). Model checking consistency of uml diagrams using alloy. In *World Academy of Science, Engineering and Technology*, volume 71, pages 547–550. Springer.

[OMG, 2003] OMG (2003). MDA Guide Version 1.0.1. Technical Report , http://www.omg.org/mda.

[OMG, 2005] OMG (2005). Uml profile for schedulability, performance and time specification.

[OMG, 2006] OMG (2006). Object constraint language : Omg available specification. Technical report, http://www.omg.org/docs/formal/06-05-01.pdf.

[OMG, 2011a] OMG (2011a). OMG Unified Modeling Language: Superstructure, V2.4.1 . Technical Report , Object Management Group.

[OMG, 2011b] OMG (2011b). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. Document formal/11-06-02, available from www.omg.org.

[Orejas et al., 2010] Orejas, F., Ehrig, H., Klein, M., Padberg, J., Pino, E., and Pérez, S. (2010). A generic approach to connector architectures part ii:

Instantiation to petri nets and csp. *Journal of Fundamenta Informaticae*, 99(1):95–124.

[Orejas et al., 2009] Orejas, F., Guerra, E., de Lara, J., and Ehrig, H. (2009). Correctness, completeness and termination of pattern-based model-to-model transformation. In *Proceedings of the 3rd Conference on Algebra and Coalgebra in Computer Science (CALCO'09)*, LNCS 5728, pages 383–397. Springer.

[Orejas and Wirsing, 2009] Orejas, F. and Wirsing, M. (2009). On the specification and verification of model transformations. *Book Chapter: Semantics and Algebraic Specification*, 5700:140–161.

[Ouardani et al., 2006] Ouardani, A., Esteban, P., Paludetto, M., and Pascal, J. C. (2006). A meta-modeling approach for sequence diagrams to petri nets transformation within the requirements validation process. In *Proceedings of the European Simulation and Modeling Confrence, Toulouse*, pages 345–349.

[Petri, 1962] Petri, C. (1962). *Kommunikation mit automaten*. PhD thesis, University of Bonn.

[Pilone and Pitman, 2005] Pilone, D. and Pitman, N. (2005). *UML 2.0 in a Nutshell, A desktop quick reference*. O'REILLY.

[Radjenovic and Paige, 2010] Radjenovic, A. and Paige, R. (2010). Behavioural interoperability to support model-driven systems integration. In *Proceedings of the Workshop on Model-Driven Interoperability*, pages 98–107. ACM Press.

[Rafe et al., 2009] Rafe, V., Rahmani, A. T., Baresi, L., and Spoletini, P. (2009). Towards automated verification of layered graph transformation specifications. *Journal of IET Software*, 3(4):276–291.

[Reisig et al., 1985] Reisig, W., W.Brauer, G.Rozenberg, and Salomaa, A. (1985). *Petri Nets : An Introduction, EATCS Monographs on Theoretical Computer Science*, volume 4. Springer-Verlag.

[Reniers, 1998] Reniers, D. M. A. (1998). *Message Sequence Chart: Syntax and Semantics*. PhD thesis.

[Ribeiro et al.,] Ribeiro, L., Dotti, F., and Bardohl, R. A formal framework for the development of concurrent object-based systems in: Formal methods in software and systems modeling. In *Proceedings of the*.

[Ribeiro and Fernandes, 2006] Ribeiro, O. and Fernandes, J. (2006). Some rules to transform sequence diagrams into coloured petri nets. In *Proceedings of the 7th Workshop and Tutorial on Practical Use of Coloured Petri nets and the CPN Tools (CPN2006)*, pages 237–256.

[Runde et al., 2005] Runde, R., Haugen, Ø., and Stølen, K. (2005). How to transform uml neg into a useful construct. In *Proceedings of the Norsk Informatikkonferanse (NIK'05)*, pages 55–66. Tapir.

[Saeki and Kaiya, 2007] Saeki, M. and Kaiya, H. (2007). Measuring model transformation in model driven development. In *Proceedings of the CAiSE'07 Forum*, pages 77–80.

[Schürr, 1995] Schürr, A. (1995). Specification of graph translators with triple graph grammars. In *in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG'94), Herrsching*. Springer.

[Selic, 2003] Selic, B. (2003). The pragmatics of model-driven development. *Journal of IEEE Software*, 20(5):19–25.

[Sendall and Kozaczynski, 2003] Sendall, S. and Kozaczynski, W. (2003). Model transformation - the heart and soul of model-driven software development. In *IEEE Software*, volume 20, pages 42–45. IEEE.

[Sgroi et al., 2004] Sgroi, M., Kondratyev, A., Watanabe, Y., Lavagno, L., and Sangiovanni-Vincentelli, A. (2004). Synthesis of petri nets from message sequence charts specifications for protocol design. In *Proceedings of the Design, Analysis and Simulation of Distributed Systems Symposium (DASD '04)*, pages 193–199.

[Shen et al., 2008a] Shen, H., Virani, A., and Niu, J. (2008a). Formalize uml 2 sequence diagrams. In *Proceedings of the 11th IEEE International Symposium on High-Assurance Systems Engineering*, pages 437–440.

[Shen et al., 2008b] Shen, H., Virani, A., and Niu, J. (2008b). Formalize uml 2 sequence diagrams. Technical Report, University of Texas at San Antonio.

[Silva and dos Santos, 2004] Silva, J. and dos Santos, E. (2004). Applying petri nets to requirements validation. In *ABCM Symposium Series in Mechatronics*, volume 1, pages 508–517. s.

[Sommerville, 2007] Sommerville, I. (2007). *Software Engineering, 9th edition.* Addison-Wesley.

[Spivey, 1992] Spivey, J. (1992). *The Z Notation: A Reference Manual.* Prentice Hall.

[Stahl et al., 2006] Stahl, T., Volter, M., Bettin, J., Haase, A., and Helsen, S. (2006). *Model Driven Software Development : Technology, Engineering, Management.* John Wiley & Sons, Ltd.

[Stevens, 2007] Stevens, P. (2007). Bidirectional model transformations in qvt: Semantic issues and open questions. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, LNCS 4735, pages 1–15.

[Stevens, 2009] Stevens, P. (2009). A simple game-theoretic approach to check-only qvt relations. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT '09)*, pages 165 – 180. Springer-Verlag.

[Störrle, 2003a] Störrle, H. (2003a). Assert, negate and refinement in uml-2 interactions. In *Proceedings of the Workshop on Critical Systems Development with UML (CSDUML'03)*, pages 79–94.

[Störrle, 2003b] Störrle, H. (2003b). Semantics of interactions in uml 2.0. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, pages 129–136. IEEE Computer Society.

[Störrle, 2004] Störrle, H. (2004). Trace semantics of interactions in uml 2.0. Technical Report, Institut fr Informatik,Ludwig-Maximilians-Universitt Mnchen.

[Straeten et al., 2007] Straeten, R. V. D., Jonckers, V., and Mens, T. (2007). A formal approach to model refactoring and model refinement. 6(2):139–162.

[Tang et al., 2010] Tang, W., Ning, B., Xu, T., and Zhao, L. (2010). Scenario-based modeling and verification for ctcs-3 system requirement specification. In *Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET'10)*, pages 400–403.

[Tarasyuk, 1998] Tarasyuk, I. V. (1998). Place bisimulation equivalences for design of concurrent and sequential systems. *Electronic Notes in Theoretical Computer Science*, 18:191–206.

[Thomas et al., 1996] Thomas, J., Nissanke, N., and K.D.Baker (1996). A hierarchical petri net framework for the representation and analysis of assembly. *Journal of IEEE Transaction on Robotics and Automation*, 12(2):268–279.

[T.Mens and Grop, 2006] T.Mens and Grop, P. (2006). A taxonomy of model transformation. *Journal of Electronic Notes in Theoretical Computer Science*, 152:125–142.

[Tran et al., 2006] Tran, H. N., Coulette, B., and Dong, B. T. (2006). A uml-based process meta-model integrating a rigorous process patterns definition. In *Product-Focused Software Process Improvement*, volume 4034 of *LNCS*, pages 429–434. Springer Berlin / Heidelberg.

[Tribastone and Gilmore, 2008] Tribastone, M. and Gilmore, S. (2008). Automatic extraction of pepa performance models from uml activity diagrams annotated with the marte profile. In *Proceedings of the 7th international workshop on Software and performance*, pages 67–78.

[Truyen, 2006] Truyen, F. (2006). The fast guide to model driven architecture: The basics of model driven architecture (mda).

[TU-Eindhoven,] TU-Eindhoven. ExSpect tool installation and user guide. Technical Report , http://www.exspect.com/.

[Uchitel and Kramer, 2001] Uchitel, S. and Kramer, J. (2001). A workbench for synthesising behaviour models from scenarios. In *Proceedings of the 23rd*

IEEE International Conference on Software Engineering (ICSE'01), pages 188– 197. IEEE Computer Society.

[Uzam, 2004] Uzam, M. (2004). The use of petri net reduction approach for an optimal deadlock prevention policy for flexible manufacturing systems. *The International Journal of Advanced Manufacturing Technology*, 23(3-4):204–219.

[Uzam et al., 2009] Uzam, M., Ko, B., Gelen, G., and Aksebzeci, B. H. (2009). Asynchronous implementation of discrete event controllers based on safe automation petri nets. *The International Journal of Advanced Manufacturing Technology*, 41(5-6):595–612.

[Vale and Hammoudi, 2008] Vale, S. and Hammoudi, S. (2008). Context-aware model driven development by parameterized transformation. In *Proceedings of the MDISIS held with CAISE'08*, pages 121–133.

[Vale and Hammoudi, 2009] Vale, S. and Hammoudi, S. (2009). An architecture for the development of context-aware services based on mda and ontologies. In *Proceedings of the International Multi Conference of Engineers and Computer Scientists 2009, Vol I (IMECS 2009)*.

[van der Aals, 1994] van der Aals, W. (1994). Putting high-level petri nets to work in industry. *Journal of Computers in Industry*, 25:45–54.

[van der Aalst, 1993] van der Aalst, W. M. P. (1993). Interval timed coloured petri nets and their analysis. In *In Proceedings of the 14th International conference on Application and Theory of Petri Nets*, pages 453–472. Springer-Verlag.

[Vanit-Anunchai, 2010] Vanit-Anunchai, S. (2010). Modelling railway interlocking tables using coloured petri nets. In Clarke, D. and Agha, G., editors,

Coordination Models and Languages, volume 6116 of *LNCS*, pages 137–151. Springer Berlin Heidelberg.

[Vicario et al., 2009] Vicario, E., Sassoli, L., and Carnevali, L. (2009). Using stochastic state classes in quantitative evaluation of dense-time reactive systems. *IEEE Transaction on Software Engineering*, 35(5):703–719.

[web services,] web services, A. Amazon elastic compute cloud (amazon ec2)online.

[Whittle and Schumann, 2000] Whittle, J. and Schumann, J. (2000). Generating statechart designs from scenarios. In *Proceedings of the 22nd international conference on Software engineering (ICSE '00)*, pages 314–323. ACM.

[Winskel and Nielsen, 1995] Winskel, G. and Nielsen, M. (1995). Models for Concurrency. In Abramsky, S., Gabbay, D., and Maibaum, T., editors, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications.

[Winskel and Saunders-Evans, 2007] Winskel, G. and Saunders-Evans, L. (2007). Event structure spans for non-deterministic dataflow. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 175(3):109–129.

[Yang et al., 2010] Yang, N., Yu, H., Sun, H., and Qian, Z. (2010). Modeling uml sequence diagrams using extended petri nets. In *Proceedings of the International Conference on Information Science and Applications (ICISA)*, pages 1–8.

[Yatake and Aoki, 2010] Yatake, K. and Aoki, T. (2010). Automatic generation of model checking scripts based on environment modeling. In *Model Checking Software*, LNCS, pages 58–75. Springer Berlin / Heidelberg.

[Y.Yang et al., 2005] Y.Yang, Q.Tan, and Y.Xiao (2005). Verifying web services composition based on hierarchical colored petri nets. In *Proceedings of the 1st international workshop on Interoperability of heterogeneous information systems (IHIS '05)*. ACM Publication.

[Zhang et al., 2009] Zhang, T., Huang, S., and Huang, H. (2009). An operational semantics for uml rt-statechart in model checking context. In *Proceedings of the 4th International Conference on Internet Computing for Science and Engineering*, pages 12–18. IEEE Computer Society.

[Zimmermann, 2008] Zimmermann, A. (2008). Restart simulation of colored stochastic petri nets. In *Proceedings of the 7th Int. Workshop on Rare Event Simulation (RESIM 2008)*, pages 143–152.