

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

A FRAMEWORK FOR TRANSFORMING, ANALYZING, AND REALIZING
SOFTWARE DESIGNS IN UNIFIED MODELING LANGUAGE

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Zhijiang Dong

2006

UMI Number: 3235930

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 3235930

Copyright 2006 by ProQuest Information and Learning Company.


All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

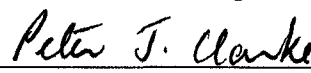
To: Dean Vish Prasad
College of Engineering and Computing

This dissertation, written by Zhijiang Dong, and entitled A Framework for Transforming, Analyzing, and Realizing Software Designs in Unified Modeling Language, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.



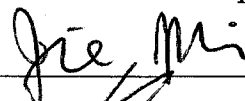
Shu-Ching Chen



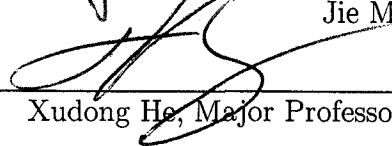
Peter J. Clarke



Yi Deng



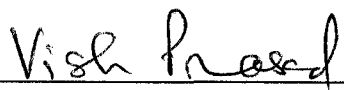
Jie Mi



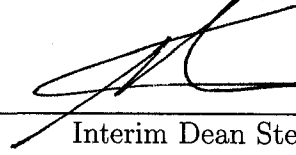
Xudong He, Major Professor

Date of Defense: July 14, 2006

The dissertation of Zhijiang Dong is approved.



Dean Vish Prasad
College of Engineering and Computing



Interim Dean Stephan L. Mintz
University Graduate School

Florida International University, 2006

DEDICATION

To my wife and son.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Xudong He, for his support, patience, and encouragement throughout my graduate studies. Without his technical and editorial advice, I would have never made this dissertation. He has taught me invaluable lessons and insights as well on the working of academic research in general. My thanks also go to the members of my committee, Shu-Ching Chen, Peter J Clarke, Yi Deng, Raimund K. Ege, and Jie Mi, for their guidance, assistance, encouragement and suggestions.

I am also grateful to my colleagues, Gonzalo Argote, Zengfai Dai, Junhua Ding, Shu Gao, Ying Huang, Lian Mo, Huiqun Yu, Tianjun Shi, Weixiang Sun, for their help and suggestions concerning both research and life in Miami during past several years. The friendship of them makes the research wonderful and life colorful.

Last but not least, I also wish to thank my wife, Yujian Fu. Mere words are not enough to express the debt of gratitude and respect I have for her. She took on much more than her share of the burden to enable me complete this effort. My parents receive my deepest gratitude and love for their dedication and many years of support leading to this dissertation.

This work is supported in part by NSF under grant HRD-0317692, and by NASA under grant NAG 2-1440. I also appreciate the financial support from the University Graduate School, Florida International University in the form of Dissertation Year Fellowship.

ABSTRACT OF THE DISSERTATION
A FRAMEWORK FOR TRANSFORMING, ANALYZING, AND REALIZING
SOFTWARE DESIGNS IN UNIFIED MODELING LANGUAGE

by

Zhijiang Dong

Florida International University

Miami, Florida

Professor Xudong He, Major Professor

Unified Modeling Language (UML) is the most comprehensive and widely accepted object-oriented modeling language due to its multi-paradigm modeling capabilities and easy to use graphical notations, with strong international organizational support and industrial production quality tool support. However, there is a lack of precise definition of the semantics of individual UML notations as well as the relationships among multiple UML models, which often introduces incomplete and inconsistent problems for software designs in UML, especially for complex systems. Furthermore, there is a lack of methodologies to ensure a correct implementation from a given UML design. The purpose of this investigation is to verify and validate software designs in UML, and to provide dependability assurance for the realization of a UML design.

In my research, an approach is proposed to transform UML diagrams into a semantic domain, which is a formal component-based framework. The framework I proposed consists of components and interactions through message passing, which are modeled by two-layer algebraic high-level nets and transformation rules respectively. In the transformation approach, class diagrams, state machine diagrams and activity diagrams are transformed into component models, and transformation rules are extracted from interaction diagrams. By applying transformation rules to component models, a (sub)system model of one or more scenarios can be constructed. Various techniques such as model checking, Petri net analysis techniques can be adopted to check if UML designs are complete or consistent. A new component called property

parser was developed and merged into the tool SAM Parser, which realize (sub)system models automatically. The property parser generates and weaves runtime monitoring code into system implementations automatically for dependability assurance. The framework in the investigation is creative and flexible since it not only can be explored to verify and validate UML designs, but also provides an approach to build models for various scenarios. As a result of my research, several kinds of previous ignored behavioral inconsistencies can be detected.

TABLE OF CONTENTS

CHAPTER	PAGE
1 Introduction	1
1.1 Problem	1
1.2 Approach	3
1.3 Benefits	5
1.4 Assumptions and Scopes	5
1.5 Thesis Overview	7
2 Theoretic Foundation	9
2.1 Introduction	9
2.2 Petri Nets	10
2.2.1 Place/Transition Nets	10
2.2.2 Algebraic High Level Nets	12
2.3 Category Theory	15
2.4 Algebraic High-Level Net Transformation Systems	19
2.5 Summary	23
3 Component-based System Modeling Framework	24
3.1 Introduction	24
3.2 Related Work	25
3.3 Informal Introduction to the Framework	27
3.4 Component Models	32
3.4.1 Function Nets	32
3.4.2 Component Nets	35
3.5 Transformation Rules	40
3.5.1 Refinement Rules	41
3.5.2 Creation Rules	43
3.5.3 Destruction Rules	44
3.5.4 Interaction Rules	46
3.5.5 Creation/Destruction Message Passing Rules	47
3.6 Component Composition	49
3.7 Analysis	49
3.7.1 Function nets	50
3.7.2 System nets	51
3.8 Summary	57
4 Verification and Validation of UML Designs	59
4.1 Introduction	59
4.2 Related Works	60
4.2.1 Formalization of UML Diagrams	60
4.2.2 Inconsistency Detection	64
4.3 Running Example	66
4.4 Algebraic View of UML Class Diagrams	66
4.5 Formalization of State Machine	69

4.6	Formalization of Activity Diagrams	71
4.7	Transformation Rules From Interaction Diagrams	74
4.8	Model Inconsistency	75
4.9	Summary	79
5	Implementation and Verification of SAM Architecture Designs	81
5.1	Introduction	81
5.2	Related Works	82
5.3	Software Architecture Model	85
5.3.1	SAM	85
5.3.2	An Example of SAM	85
5.4	Methodology	89
5.5	Implementation of Petri Nets	91
5.6	Implementation of Run-time Verification	93
5.7	Experimental Results	98
5.8	Summary	101
6	Conclusion	103
6.1	Overview	103
6.2	Contributions	104
6.3	Future Work	105
	LIST OF REFERENCES	107
	APPENDICES	121
	VITA	149

LIST OF TABLES

TABLE		PAGE
1	Benefits of Dissertation Research	5
2	Transition Firing Sequence of <i>Watson</i> Joining Table	33
3	Summary of Production Types	41
4	Generated Files for Coffee Machine	99
5	State Machine Diagram Formalization Rules	133

LIST OF FIGURES

FIGURE		PAGE
1	Overview of Investigation Approach	4
2	Scope: a Subset of UML	6
3	Petri Net of Consumer-Producer System	12
4	Algebraic High-Level Net of Consumer-Producer System	15
5	Component Models in Hurried Philosopher Example	29
6	A Transformation Example	31
7	System Net of Dining Philosopher Example	32
8	The Semantics of a Component	39
9	A Refinement Production	42
10	A Creation Production	43
11	A Destruction Production	44
12	An Interaction Production	47
13	A Creation Message Passing Production	48
14	The Valid Function Net of the Servant	51
15	A Simple Online Shopping System	67
16	UML Formalization Pattern	68
17	Petri Net Representation of Actions	73
18	Transformation Rule for Passing Message <i>AddItem</i>	75
19	SAM Parser Overview	82
20	A SAM Architecture Model	86
21	SAM Model of Coffee Machine	87
22	Behavior of Subcomponents in CoffeeMachine	88
23	Framework of SAM Parser	90
24	Petri nets satisfying or violating guidance	93
25	The Formal Net of a Simple State	134
26	The Formal Net of Composite State	135

27 The Formalization of Initial State 137

28 The Formalization of a Simple Transition 139

29 The Formalization of a Cross Transition(Leaving) 141

30 The Formalization of a Cross Transition(Entering) 142

31 The Formalization of a Group Transition(leaving) 144

32 The Formalization of a Compound Transition(Join) 146

CHAPTER 1

INTRODUCTION

1.1 Problem

Modeling languages play a critical role in software development process. One of the major functionalities of modeling languages is to provide a complete and valid system model based on which various techniques such as model checking, theorem proving, and refinement are applied to improve quality and efficiency in terms of cost and time. Last several decades have witnessed the emergence of more than 50 modeling languages [51]. Currently, Unified Modeling Language (UML for short) [120] is the most comprehensive and accepted object-oriented, multi-paradigm modeling language. UML supports the multi-view approach, i.e. artifacts created in the development process for various views are modeled by various kinds of UML concepts. More specifically, class diagrams specify static structure of systems; statechart diagrams describe behavior of individual classifier; activity diagrams emphasize control flows and object flows for coordinating low-layer behaviors, rather than which classifier owns these behaviors; interaction diagrams including sequence diagrams and communication diagrams realize use cases by describing interactions of objects to complete a task.

Generally speaking, UML designs capture system requirements, establish abstract models, and serve as the corner stone for system implementation. Therefore, software quality, costs, adherence to schedule largely depends on the “quality” of UML designs we build. More specifically UML designs should meet these characteristics:

completeness, validness, and consistency. Completeness indicates that all important system aspects should have been captured before entering the next phrase. Validness means UML designs should satisfy expected system properties that are not specified by UML designs. Consistency implies there is no conflict information among UML designs. Unfortunately such goal is hard to achieve due to characteristic of modeling languages as well as characteristics of UML.

First, the lack of precise semantics hinders further analysis, and brings misunderstanding of models, which cause errors in the final system model. System requirements and models should be easy to understand, not only for developers, but also for clients and end users who generally have little knowledge of modeling languages and software engineering. So modeling languages are generally informal languages that lack precise and unambiguous semantics. In other words, it is possible that peoples such as clients, developers and designers may have different, even conflict understanding for the same concept, artifact, or model. Although UML provides a good balance between understandability and formal syntax, its semantics is defined by plain natural language, which is in general ambiguous, and confusing.

Second, inconsistency is introduced by the multi-view and multi-notation approach. UML supports the multi-view and multi-notation approach, which helps designers focus on individual viewpoint so that the models are more manageable and less error-prone. However, inconsistencies arise because “the models overlap – that is they incorporate elements which refer to common aspects of the system under development – and make assertions about these aspects which are not jointly satisfiable as they stand, or under certain conditions” [147]. The detection of inconsistencies is not easy due to the multi-notations.

Third, system complexity, project pressure of cost and schedule may ignore important aspects and scenario, and may introduce undetected errors as well as conflict information. With the progress of software development technology, systems to be

built are becoming more and more complex, and more and more people acting as different roles are involved in system development process. With heavy time pressure to market and limited resource, there are more chances to establish a poor system model in terms of undetected errors. Even worse, some important aspects and scenario may be ignored in the final model since they are originally thought as trivial and there is no time or cost to model these “trivial” aspects.

Fourth, system requirements from which system models are built may contain conflict information since requirements from stakeholders of different interests are related, and even on opponent sides.

All above matters make it hard to build a valid, complete, and consistent UML designs. In this investigation, I proposed an approach to verify and validate UML designs. Since a “correct” UML design does not guarantee a “correct” system implementation due to the error-prone realization process, a tool was developed for dependability assurance to generate runtime monitor code to verify system properties during program execution.

1.2 Approach

The approach to verify and validate UML designs is portrayed in Fig. 1.

The core part of the approach is the proposed component-based framework, which can be explored to model systems consisting of components that interact with each other through message passing. This framework provides a formal way to model components and interactions in Petri nets and transformation rules respectively. A whole (sub)system model can be constructed by integrating component models together according to interaction models, just like the assembly of numerous small interlocking and tessellating pieces to produce a complete picture.

Given UML designs of a system, we can transform various UML diagrams, i.e. class diagrams, state machine diagrams, activity diagrams and interaction diagrams

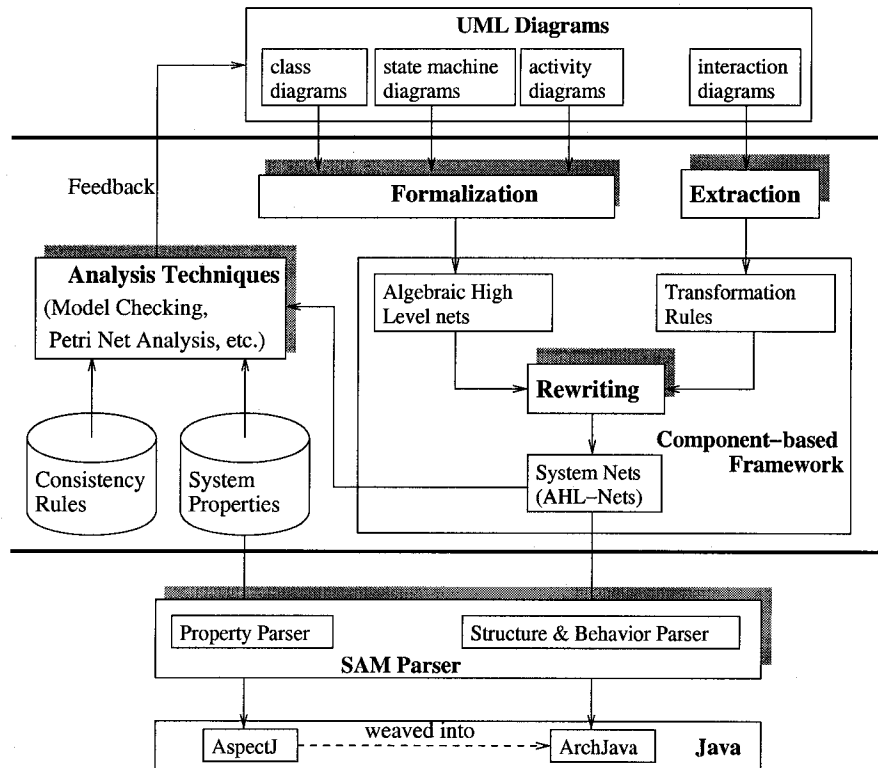


Figure 1: Overview of Investigation Approach

into corresponding parts of the framework according to given rules. More specifically, class diagrams, state machine diagrams, and activity diagrams are formalized and integrated into component models, and transformation rules can be extracted from interaction diagrams to specify the possible message passing between various component models.

Although various analysis techniques such as model checking, theorem proving, and Petri net analysis techniques can be explored to analyze system models constructed in the framework, we chose model checking to verify system models against system properties and detect inconsistency among UML designs. Petri net analysis techniques maybe used as a complement to detect some specific inconsistencies.

A correct and consistent UML design cannot guarantee a complete and correct system realization because of the informal and error-prone implementation process. A component – **Property Parser** was developed and plugged into the tool **SAM**

Parser for dependability assurance that automatically realize system models constructed from the framework. For given properties, **Property Parser** generates runtime monitor code automatically, which is weaved into functionality code through aspect-oriented programming. Therefore, properties can be verified during program execution.

The remainder of this dissertation will trace each part of Fig. 1, demonstrating how to verify and validate UML designs and how to generate and weave runtime monitor code for dependability assurance.

1.3 Benefits

The contributions and benefits that follow from this investigation are enumerated in Table 1, along with an explanation of how each of them are realized. The summary chapter 6 gives a much more detailed explanation of each benefit or contribution and how each was realized in the dissertation.

Table 1: Benefits of Dissertation Research

	Benefit	Explanation
1	Development of a formal component-based framework to model systems	Components and their interactions are modeled by Petri nets and transformation rules, respectively. The (sub)system model can be constructed by applying transformation rules to components according to analysis needs.
2	Formalization of UML diagrams	Class diagrams and state machine diagrams are formalized by algebraic specifications and Petri nets, respectively.
3	Development of a process to integrate UML designs into a system model	Application of the proposed framework to UML designs
4	Development of a process to validate and verify UML designs	Model checking and other Petri net analysis techniques are explored to analyze system models obtained from UML designs.
5	Development of a tool to implement system models and validate the implementation automatically	Incorporation of runtime verification technique and aspect-oriented programming in the tool

1.4 Assumptions and Scopes

It is assumed that:

1. This investigation is limited to the UML 2.0.
2. UML supporting CASE tools check initial specification consistency (within an individual diagram) and compliance to UML syntax. The application of my work on invalid UML specifications would be unpredictable.
3. This investigation only focuses on a subset of UML depicted in Fig. 2. Since the investigation becomes too complicated when UML diagrams are used in broad situations as indicated in UML whitebook [120], we only consider the most popular usage of involved diagrams, which are summarized as the following:

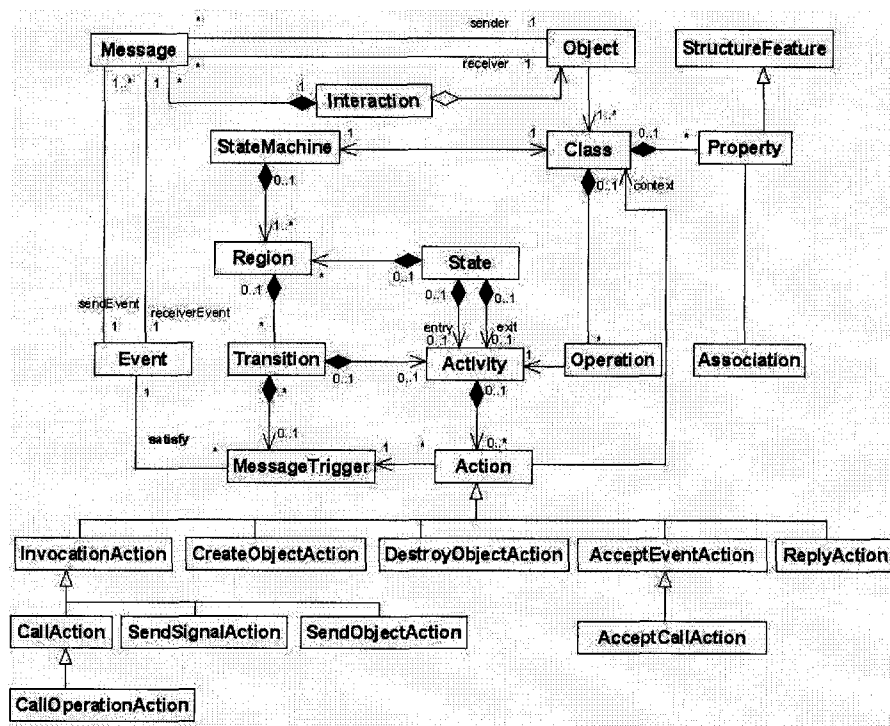


Figure 2: Scope: a Subset of UML

- Each non-primitive (see section 4.4) class operation is described by an activity;

- The actions in activities are restricted to those actions as showed in the figure. Other actions such as link related actions and variable related actions are represented as invocation actions due to the formalization of class diagrams based on algebraic specifications. Some other actions such as *ReadExtentAction*, *RaiseException* are just ignored in the current work.
- Only *MessageTrigger* is allowed in current work. Other triggers such as *TimeTrigger* and *ChangeTrigger* are not considered.
- Time and timing concepts in interaction diagrams are just ignored in the current work since they are out of modeling power of non-timed Petri nets.

Additionally, we assume that each object in interaction diagrams refers to a class declared in the class diagram. There is a statechart diagram for each class to describe its behavior. Furthermore, each activity in statechart diagrams is specified by an activity diagram.

4. There are different kinds of inconsistencies [94, 95]: horizontal v.s. vertical inconsistency, inter- v.s. intra-inconsistency, and syntactic v.s. semantic inconsistency. My investigation is limited to horizontal, semantic, and inter-inconsistency.

1.5 Thesis Overview

Currently, UML is the most popular object oriented modeling language. As a multi-paradigm language, UML can enjoy the benefits by modeling various system aspects in different UML diagrams. Unfortunately, UML designs also inherit the inconsistency problems – multiple UML designs may contain conflict information. Things are even worse since a correct and consistent UML design does not guarantee a valid system implementation. My dissertation describes the proposed framework through which UML designs are validated and verified, and the approach to generate and weave runtime monitor code automatically for dependability assurance.

Chapter 2 briefly illustrates Algebraic High Level nets, category theory, and transformation system as a background to understanding the proposed framework. The related works are distributed to the following three chapters to cover specific topics.

The component-based system modeling framework is given in Chapter 3. The framework consists of several parts: the way to model components and interactions, and the approach to compose system models by applying transformation rules.

Chapter 4 shows the approach to verify and validate UML designs by exploring the framework described in Chapter 3. More specifically, component models are constructed from class diagrams, state machine diagrams, and activity diagrams, while transformation rules are extracted from interaction diagrams. Then different analysis techniques can be adopted to analyze the system net that are constructed by applying transformation rules.

The automated system implementation from system models is given in Chapter 5. The tool presented in this chapter can be used to implement automatically system models constructed in Chapter 4, and more importantly, to validate the implementation by adopting runtime verification technique and aspect-oriented programming.

Chapter 6 contains the conclusions from this investigation and provides recommendation for future research.

CHAPTER 2

THEORETIC FOUNDATION

2.1 Introduction

It was stated in Chapter 1 that the proposed framework integrates different theories seamlessly, i.e. Petri nets, category theory, and graph transformation.

Petri nets [112], introduced by Dr. Carl Adam Petri in his PhD thesis (Kommunikation mit Automaten), is a formal and graphical appealing language that is appropriate for modeling concurrent and distributed systems. A main motivation for the use of Petri nets in concurrent and distributed systems modeling is the possibility to formally state and decide certain desirable system properties, such as liveness and boundedness. There are in general two kinds of Petri nets: low-level Petri nets and high-level Petri nets. Although they have the same expressive power, high level Petri nets provide a more succinct and manageable system description.

Category theory [26] deals in an abstract way with mathematical structures and relationships among them. Categories are an abstract mathematical construct consisting of category objects and category arrows. In general, category objects are the objects in the category of interest while category arrows define a morphism from the internal structure of one category object to another. Instead of focusing merely on the individual objects possessing a given structure, as mathematical theories have traditionally done, category theory emphasizes the morphisms – the structure-preserving processes – between these objects. In this research, category objects of interests are

algebraic specifications, Petri nets, and category arrows are specification morphisms and Petri net morphisms.

In graph theory, graph transformation/rewriting is a system of rewriting for graphs. During the application of graph rewriting to a graph, subgraphs are replaced according to the rules of a rewrite system. There are several approaches to graph rewriting, one of them is the algebraic approach, which is based upon category theory. Actually the algebraic approach is divided into at least three sub approaches: the double-pushout approach (DPO), the single-pushout approach (SPO) and (more recently) the pullback approach. In this research, DPO approach is chosen to change Petri nets due to the strong constraints on applying rules to rewrite graphs.

In this chapter, Section 2 gives a brief introduction of Petri nets, including Place/Transition Nets and algebraic high-level nets. Category theory and algebraic high-level net transformation systems are illustrated in Sections 3 and 4 respectively.

2.2 Petri Nets

In this section, I introduce two kinds of Petri nets: Place/Transition nets (a variant of low level Petri nets) and algebraic high-level nets (a variant of high level Petri nets).

2.2.1 Place/Transition Nets

Definition 1 (Place/Transition Nets) *A place/transition net is a 5-tuple $\langle P, T, F, W, M_0 \rangle$, where*

- *P is a finite and non-empty set of places,*
- *T is a finite and non-empty set of transitions disjoint from P , i.e. $P \cap T = \emptyset$,*
- *F is the set of arcs, $F \subseteq (P \times T) \cup (T \times P)$,*
- *W is the arc weight function, $W : F \longrightarrow \mathbb{N}$,*
- *$M_0 \in \mathbb{M}$ is the initial marking, where \mathbb{M} is the set of markings, $\mathbb{M} = \{M : P \longrightarrow \mathbb{N}\}$.*

Places, transitions, and arcs are the basic structures in Petri nets. Places model system status; transitions indicate actions a system may take; while arcs illustrate data flows as well as control flows. A place can contain data called tokens. A marking of a Place/Transition net is a distribution of tokens over all places. The initial marking M_0 defines the initial system status.

For convenience, we introduce symbols $\bullet p$ (p^\bullet , respectively) for a place $p \in P$ to illustrate the set of transitions t such that $(t, p) \in F$ ($(p, t) \in F$, respectively). Similarly, we can define $\bullet t$ (t^\bullet , respectively) for a transition $t \in T$.

Petri nets are executable. More specifically, transitions act on input tokens by a process known as firing. A transition is enabled if it can fire, i.e., there are enough tokens in every input place. When a transition fires, it consumes tokens from input places, performs some processing task, and places a specified number of tokens into each output place. It does this atomically, i.e. in one single non-preemptible step. This is the dynamic semantics of Petri nets, which are specified formally by the following definitions.

Definition 2 *Let $\langle P, T, F, W, M_0 \rangle$ be a Place/Transition net. A transition is enabled at a marking M if and only if (iff for short): $\forall p \in \bullet t : M(p) \geq W(p, t)$.*

A transition t leads (can be fired) from a marking M to a marking M' ($M[t > M'$ for short) iff t is enabled at M and: $\forall p \in P : M'(p) = M(p) - W(p, t) + W(t, p)$.

Execution of Petri nets is nondeterministic. In other words, multiple transitions can be enabled at the same time, and any one of which can fire. This characteristic makes Petri nets suitable for modeling concurrent behavior of distributed systems.

Figure 3 shows a Place/Transition net for a consumer and producer system. The places, transitions, arcs are denoted by circles, rectangles, and arrows, respectively. A dot indicates a token in a specific place. The weight of a arc is described by the integer along an arrow (1 by default). The initial marking is (idle=1, ready=0, storage=0, accepted=0, waiting

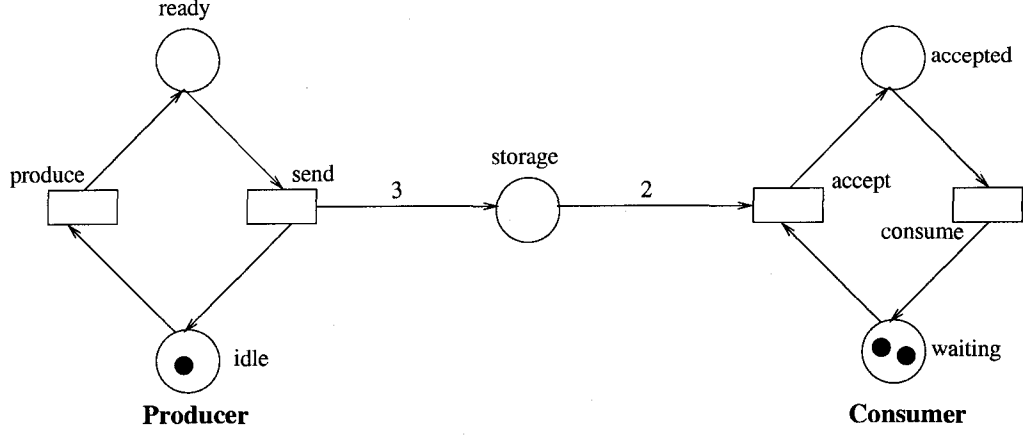


Figure 3: Petri Net of Consumer-Producer System

$=2)$ (abbreviated as $(1,0,0,0,2)$). Then the following is a firing sequence:
 $(1,0,0,0,2)[\text{produce}]>(0,1,0,0,2)[\text{send}]>(1,0,3,0,2)[\text{accept}]>(1,0,1,1,1)[\text{consume}]>(1,0,1,0,2)$.

2.2.2 Algebraic High Level Nets

High level Petri nets extend the basic Place/Transition net formalism by distinguishing tokens. More specifically the values of tokens in high level Petri nets are typed. Algebraic high level nets [47], a variant of high level Petri nets, use algebra to define token types. This section is intended to introduce basic concepts of signature, algebraic specification, algebra and Algebraic high level nets.

Given a set P , the free commutative monoid $(P^\oplus, \lambda, \oplus)$ is generated by P such that λ is the neutral elements and the binary operation \oplus satisfies associativity and commutativity. Elements ω of the free commutative monoid P^\oplus over some set P can be represented as $\omega = \sum_{p \in P} (c_p \cdot p)$ with coefficients $c_p \in \mathbb{N}$. They can be considered as multi-sets. In the following, we let λ be the empty multi-set, and define binary operation \oplus as $\omega_1 \oplus \omega_2 = \sum_{p \in P} ((c_p + d_p) \cdot p)$. The inverse operation \ominus of \oplus is defined as $\omega_1 \ominus \omega_2 = \sum_{p \in P} ((c_p - d_p) \cdot p)$ if $\omega_2 \leq \omega_1$, i.e. for any $p \in P$, $d_p \leq c_p$.

A signature $SIG = (S, OP)$ consists of a set S of sorts, and a set OP of constant and operation symbols. Each operation symbol O is indexed by a pair (σ, s) , $\sigma \in S^*$ and $s \in S$ denoted by $O_{\sigma, s}$. σ is called the argument sorts and s the range sort of

operator o . Let X_s be a finite set of variables of sort s . $X = \bigcup_{s \in S} X_s$ is the finite set of variables w.r.t. the signature SIG . The set $T_{OP,s}(X)$ of terms of sort s is inductively defined by:

- $X_s \subseteq T_{OP,s}(X)$;
- $O(t_1, \dots, t_n) \in T_{OP,s}(X)$ for all operation symbol $O \in OP$ with $O : s_1 \dots s_n \rightarrow s$ and all terms $t_1 \in T_{OP,s_1}(X), \dots, t_n \in T_{OP,s_n}(X)$.

For convenience, we introduce symbol $T_{OP}(X) = \bigcup_{s \in S} T_{OP,s}(X)$ to denote the set of all terms, and symbol $T_{OP,s} = T_{OP,s}(\emptyset)$ to denote the set of terms not containing variables (also called ground terms).

A **SIG**-algebra $A = (S_A, OP_A)$, providing an interpretation for a signature $SIG = (S, OP)$, consists of two families $S_A = (A_s)_{s \in S}$ and $OP_A = (O_A)_{O \in OP}$ where A_s are sets for all $s \in S$, called domain of A , and $O_A : A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$ is a function for each operation symbol $O : s_1 \times \dots \times s_n \rightarrow s$. Given an assignment $ass : X \rightarrow A$ with $ass(x) \in A_s$ where $x \in X_s$ and $s \in S$. The extended assignment, or simply extension $\overline{ass} : T_{OP}(X)^\oplus \rightarrow A^\oplus$ of the assignment ass is recursively defined by:

- $\overline{ass}(x) = ass(x)$ for all variables $x \in X$;
- $\overline{ass}(o(t_1, \dots, t_n)) = O_A(\overline{ass}(t_1), \dots, \overline{ass}(t_n))$ for all $O(t_1, \dots, t_n) \in T_{OP}(X)$.
- for any $\omega = \sum_k (c_k \cdot t_k)$ where $k, c_k \in \mathbb{N}$, and $t_k \in T_{OP}(X)$, $\overline{ass}(\omega) = \sum_k (c_k \cdot \overline{ass}(t_k))$.

An algebraic specification $SPEC = (SIG, E)$ consists of a signature SIG and a set of equations E w.r.t. the signature SIG . In the context of this paper, only positive conditional equations are considered. An **SPEC**-algebra is an **SIG**-algebra satisfying all equations in E .

Definition 3 (Algebraic High-Level Net [47]) An *algebraic high-level net* (AHL-net) N is a 9-tuple $(SPEC, X, P, T, type, cond, pre, post, A)$ where

- $SPEC = (SIG; E)$ is an algebraic specification with the signature $SIG = (S, OP)$ and a set of equations E ;
- X is a set of variables w.r.t. the specification $SPEC$;
- P is a finite set of elements called Places;
- T is a finite set of elements called Transitions disjoint from P ($P \cap T = \emptyset$);
- $type : P \rightarrow S$, assigning each place $p \in P$ a sort $type(p) \in S$;
- $cond : T \rightarrow \mathcal{P}_{fin}(EQNS(SIG; X))$, assigning each transition a finite set of equations w.r.t. the signature SIG and the set of variables X , where \mathcal{P} denotes the power set;
- $pre, post : T \rightarrow \bigoplus_{p \in P} (T_{OP, type(p)}(X) \times \{p\})^\oplus$;
- A is a **SPEC**-algebra.

Similar to Place/Transition nets, symbols $\bullet p$, p^\bullet , $\bullet t$, t^\bullet denote the set of pre- and post- transitions/places for a given place/transition, respectively. A marking of an AHL-net is denoted by $M \in \{(a, p) \mid a \in A_{type(p)}, p \in P\}^\oplus$. Let $\alpha: Var(t) \rightarrow A$ be a variable assignment where $Var(t)$ is the set of variables occurred in $cond(t)$, $pre(t)$ and $post(t)$ for any transition $t \in T$. Transition t is enabled with the binding α under the marking M if the transition condition $cond(t)$ is validated in A under function $\bar{\alpha}$ and $\bar{\alpha}(pre(t)) \leq M$. Then the marking $M' = M \ominus \bar{\alpha}(pre(t)) \oplus \bar{\alpha}(post(t))$ is computed by firing the transition t with the binding α under the marking M .

Figure 4 [151] shows an AHLN for a consumer and producer system. The algebraic specification declares 4 types (nat, bool, data, and queue), 2 constants (err of data type, nil of queue type), and 5 operations. Each place has an associated type, and each transition has a set of equations. By default, the equations in a transition like co always hold. The “weight” associated with an arc is a multi-set of terms defined in the related algebra.

```

sorts: nat, bool, data, queue
opns: err: data, nil: queue
      inq: data queue queue
      deq: queue queue
      first: queue data
      empty: queue bool
      length: queue nat
eqns: deq(nil)=nil
      deq(inq(x,nil))=nil
      deq(inq(x,inq(y,q)))=inq(x,deq(y,q))
      first(nil)=err
      first(inq(x,nil))=x
      first(inq(x,inq(y,q)))=first(inq(y,q))
      empty(nil)=true
      empty(inq(x,q))=false
      length(nil)=0
      length(inq(x,q))=length(q)+1

```

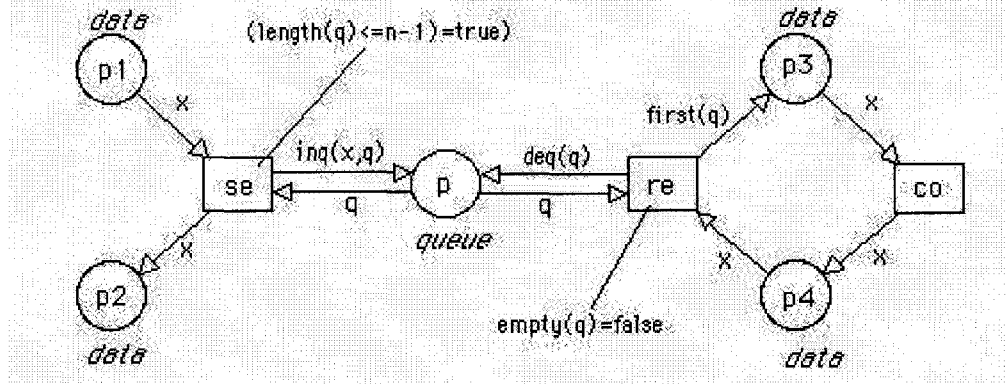


Figure 4: Algebraic High-Level Net of Consumer-Producer System

2.3 Category Theory

All definitions in this section are from [26].

Definition 4 (Category) A category φ consists of a class $|\varphi|$ (whose elements are called objects of the category), and a class of arrows between any two objects (called morphism), which satisfies following conditions:

1. Morphism Composition: $(A \rightarrow B) \circ (B \rightarrow C) = (A \rightarrow C)$;
2. Identity morphism $1_A \in \varphi(A, A)$ exists for any object A ;
3. Associativity axiom: given morphisms $f : A \rightarrow B, g : B \rightarrow C, h : C \rightarrow D$, then $h \circ (g \circ f) = (h \circ g) \circ f$.
4. Identity axiom: given morphisms $f : A \rightarrow B, g : B \rightarrow C$, then $1_B \circ f = f, g \circ 1_A = g$.

Example 1 The category **SPEC** consists of algebraic specifications (S, OP, E) and of specification morphisms $f = (f_S : S1 \rightarrow S2, f_{OP} : OP1 \rightarrow OP2) : SPEC1 \rightarrow SPEC2$ satisfying $f(o : s_1 \dots s_n \rightarrow s) = f_{OP}(O) : f_S(s_1) \dots f_S(s_n) \rightarrow f_S(s)$ and such that $f^\#(E1) \subseteq E2$ where $f^\#$ is the unique extension of f to terms and equations [45]. Specification morphism f is injective if functions f_S and f_{OP} are injective. Specification morphism f is strict if, given an arbitrary positive conditional equation e , we have $f^\#(e) \in E2$, then $e \in E1$.

Definition 5 (Functor) A functor F from a category **SPEC1** to a category **SPEC2** is a mapping, which maps a object, a morphism of the category **SPEC1** to a object, a morphism of the category **SPEC2** respectively and satisfies following conditions:

1. $F(A1 \rightarrow A2)$ is a morphism from $F(A1)$ to $F(A2)$ of the category **SPEC2** ;
2. For every pair of morphisms $f : A \rightarrow A'$ and $g : A' \rightarrow A''$: $F(g \circ f) = F(g) \circ F(f)$;
3. For every object A of the category **SPEC1** : $F(1_A) = 1_{F(A)}$;

There is a special kind of functor, called forgetful functor, which leaves the objects and the arrows as they are, but forget the extra structure or algebraic properties. Let $f : SPEC1 \rightarrow SPEC2$ be a specification morphism of category **SPEC** where $SPEC_i = (S_i, OP_i, E_i)$ for $i = 1, 2$. The corresponding forgetful functor $V_{fSPEC} : CAT(SPEC2) \rightarrow CAT(SPEC1)$ is defined as $V_{fSPEC}(A2) = A1$ where $A1$ is an object of the category of all **SPEC1**-algebras denoted by $CAT(SPEC1)$ such that:

$$A1_s = A2_{f_S(s)} \quad \text{for all } s \in S_1$$

$$O_{A1} = f_{OP}(O)_{A2} \quad \text{for all } O \in OP_1$$

for all SIG2-homomorphism $h' : A_2 \rightarrow B_2 : V_{f_{SPEC}}(h') = h : A_1 \rightarrow B_1$ with

$$h_s = h'_{f_S(s)} \quad \text{for all } s \in S_1$$

With the forgetful functor, we can define the category of algebraic high-level nets. Let A_i be **SPEC** $_i$ -algebras for $i=1,2$. A generalized homomorphism $F : A_1 \rightarrow A_2$ consists of a pair $f = (f_{SPEC}, f_A)$ where f_{SPEC} is a morphism of category **SPEC** and $f_A : A_1 \rightarrow V_{f_{SPEC}}(A_2)$ is a **SPEC** $_1$ -homomorphism. Composition of generalized homomorphisms $f = (f_{SPEC}, f_A) : A_1 \rightarrow A_2$ and $g = (g_{SPEC}, g_A) : A_2 \rightarrow A_3$ is given by: $g \circ f = (g_{SPEC} \circ f_{SPEC}, V_{f_{SPEC}}(g_A) \circ f_A) : A_1 \rightarrow A_3$.

The category **AHLNET** of algebraic high-level nets consists of all AHL-nets N as objects and quadruples $f = (f_{SPEC}, f_P, f_T, f_A)$ as morphisms where

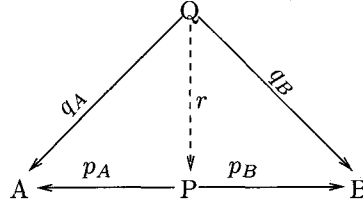
- $f_{SPEC} : (S_1, OP_1, E_1) \rightarrow (S_2, OP_2, E_2)$ is a specification morphism of category **SPEC** ;
- $f_T : T_1 \rightarrow T_2$ and $f_P : P_1 \rightarrow P_2$ are functions;
- $(f_{SPEC}, f_A) : A_1 \rightarrow A_2$ is a generalized homomorphism and $f_A : A_1 \rightarrow V_{f_{SPEC}}(A_2)$ is an isomorphism in **CAT**(**SPEC** $_1$).

such that the following diagram commutes componentwise.

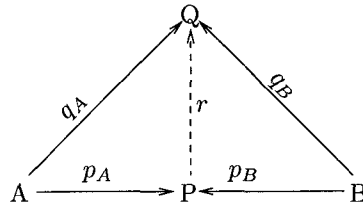
$$\begin{array}{ccccc}
 \mathcal{P}_{fin}(EQNS(SIG1)) & \xleftarrow{cond1} & T1 & \xrightleftharpoons[post1]{pre1} & (T_{OP1}(X1) \times P1)^{ab} \\
 \downarrow \mathcal{P}_{fin}(f_{SIG}^\#) & & \downarrow f_T & = & \downarrow (f_{SIG}^\# \times f_P)^{ab} \\
 \mathcal{P}_{fin}(EQNS(SIG2)) & \xleftarrow{cond2} & T2 & \xrightleftharpoons[post2]{pre2} & (T_{OP2}(X2) \times P2)^{ab}
 \end{array}$$

Definition 6 (Products) Let **CAT** be a category and A, B two objects of **CAT**. A product of A and B is, by definition, a triple (P, p_A, p_B) where P is an object of

CAT, and $p_A : P \rightarrow A$ and $p_B : P \rightarrow B$ are morphisms, and for any other similar triple $(Q, q_A : Q \rightarrow A, q_B : Q \rightarrow B)$ there exists a unique morphism $r : Q \rightarrow P$ such that $q_A = p_A \circ r$ and $q_B = p_B \circ r$.

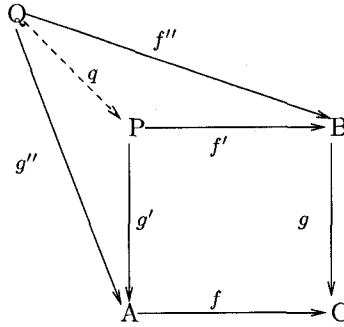


Definition 7 (CoProducts) Let **CAT** be a category and A, B two objects of **CAT**. A product of A and B is, by definition, a triple (P, p_A, p_B) where P is an object of **CAT**, and $p_A : A \rightarrow P$ and $p_B : B \rightarrow P$ are morphisms, and for any other similar triple $(Q, q_A : A \rightarrow Q, q_B : B \rightarrow Q)$ there exists a unique morphism $r : P \rightarrow Q$ such that $q_A = r \circ p_A$ and $q_B = r \circ p_B$.

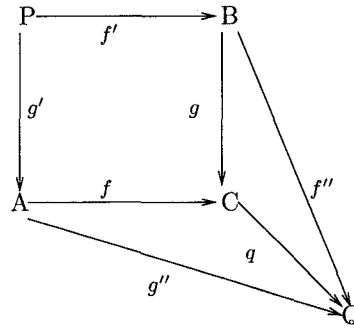


In category theory, a pullback is the limit of a diagram consisting of two morphisms with a common codomain. The duo notation of pullback is that of pushout, just like the relationship between product and coproduct. The form definition of pullback and pushout are given in the following.

Definition 8 (Pullback) Consider two morphisms $f : A \rightarrow C$ and $g : B \rightarrow C$ in a category **CAT**. A pullback of (f, g) is a triple (P, f', g') such that P is an object of **CAT** and $f' : P \rightarrow B$, $g' : P \rightarrow A$ are morphisms of **CAT** satisfying $f \circ g' = g \circ f'$; and for every other similar triple (Q, f'', g'') , there exists a unique morphism $q : Q \rightarrow P$ such that $f'' = f' \circ q$ and $g'' = g' \circ q$.



Definition 9 (Pushout) Consider two morphisms $f : C \rightarrow A$ and $g : C \rightarrow B$ in a category CAT . A pushout of (f, g) is a triple (P, f', g') such that P is an object of CAT and $f' : B \rightarrow P$, $g' : A \rightarrow P$ are morphisms of CAT satisfying $g' \circ f = f' \circ g$; and for every other similar triple (Q, f'', g'') , there exists a unique morphism $q : P \rightarrow Q$ such that $f'' = q \circ f'$ and $g'' = q \circ g'$.



2.4 Algebraic High-Level Net Transformation Systems

Graphs are a very useful means to describe complex structures and systems, and to model concepts and ideas in a direct and intuitive way. These structures are often augmented by formalisms that add to the static description a further dimension modeling the evolution of systems via any kind of transformation of such graphical structures. By applying a transformation rule to replace a subgraph, the original graph is evolved into a new graph. Therefore, graph transformation can be exploited to specify the graph evolution formally.

This section is intended as an introduction to Algebraic High-Level Net Transformation Systems, a specific application of graph transformation theory to Algebraic High-Level Nets. Algebraic High-Level Net Transformation Systems were first proposed by Padberg et al. in [128]. Therefore, we adopt their concepts, symbols, and definitions in the rest of this section.

An HLR-category (\mathbf{CAT}, M) consists of a category \mathbf{CAT} together with a distinguished class M of morphisms, which is a subset of the class of morphisms in category \mathbf{CAT} . The objects in \mathbf{CAT} are called high-level structures (HL-structures for short).

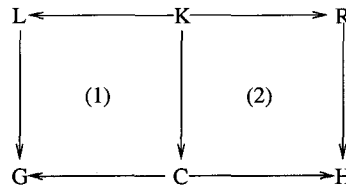
$(\mathbf{AHLNET}, M_{\mathbf{AHLN}})$ is a HLR-category where

$$M_{\mathbf{AHLN}} = \{f = (f_{\text{SPEC}}, f_P, f_T, f_A) \mid f \text{ is a morphism of } \mathbf{AHLNET}, f_{\text{SPEC}} \text{ is strict injective and } f_P, f_T \text{ injective} \}$$

$M_{\mathbf{AHLN}}$ denotes the class of morphisms used in the definition of the productions. By choosing injective morphisms, the relation of interface and left (right) side is restricted to a somehow unique way.

Definition 10 (Production and Derivation)

- A **production** $p = (L \leftarrow K \rightarrow R)$ in an HLR-category (\mathbf{CAT}, M) consists of a pair of objects (L, R) , called left- and right-hand side, respectively, an object K , called a gluing object or interface, and two morphisms $K \rightarrow L$ and $K \rightarrow R$ belonging to the class M .
- Given a production p as above and an object C together with a morphism $K \rightarrow C$. A **direct derivation** from an object G to an object H via p (written $p:G \Rightarrow H$) is given by two pushout diagrams (1) and (2) in the category \mathbf{CAT} :



The morphism $L \rightarrow G$, respectively $R \rightarrow H$ are called occurrence of L in G , respectively, R in H . C is called the pushout complement.

- A derivation sequence $G \Rightarrow^* H$ from G to H is either $G \cong H$ (isomorphism), or a sequence of $n \geq 1$ direct derivations: $G = G_0 \Rightarrow G_1 \Rightarrow \dots \Rightarrow G_n = H$ via (p_1, \dots, p_n) .

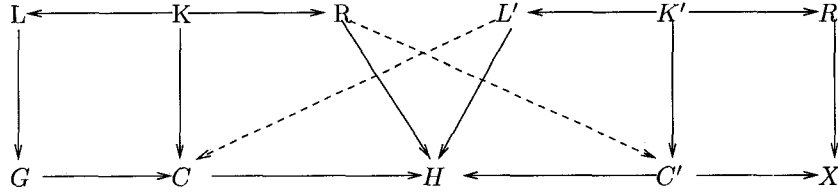
The gluing condition is introduced to construct pushout complement in order to achieve a constructive view. More specifically, the gluing condition states how to delete some part while still obtaining a well-defined HL-structure as pushout complement. Due to the space limit, we cannot give the gluing condition for the category **AHLNET**. More detailed information of gluing condition and construction of pushout complement can be found in [128]. We just want to point out that the gluing condition for **AHLNET** is equivalent to a pushout of **AHLNET**. More specifically, if two morphisms $f:K \rightarrow L$ and $g:L \rightarrow G$ of category **AHLNET** with $f \in M_{AHLN}$ meet the gluing condition, then the pushout complement C exists. On the other hand, if the diagram (1) in the definition of production is a pushout such that morphism $K \rightarrow L \in M_{AHLN}$, then morphism $K \rightarrow L$ and $L \rightarrow G$ satisfy the gluing condition.

Definition 11 (AHL-net Transformation System) An AHL-net transformation system $ATS = (S, \mathbb{P})$ based on an HLR-category $(\mathbf{AHLNET}, M_{AHLN})$ is given by an object S of **AHLNET**, called the initial HL-structure, a set of productions \mathbb{P} . The language of an AHL-net transformation system ATS , denoted by $L(ATS)$, is a set of AHL-nets derived from S via a sequence of productions, i.e. $L(ATS) = \{N \mid N \text{ is an AHL-net such that there is a sequence of productions } p_1, \dots, p_m \in \mathbb{P} \text{ with } S \Rightarrow N \text{ via } p_1, \dots, p_m\}$.

Our definition of AHL-net transformation system is a little different from the definition given in [128] since we do not care about terminal objects derived from the initial HL-structure. What we are interested is a subset of derived HL-structures

via some productions. In order to describe derivations over a set of productions, the following concepts are introduced.

Definition 12 (Independence) *Given two productions $p = (L \leftarrow K \rightarrow R)$ and $p' = (L' \leftarrow K' \rightarrow R')$ in an HLR-system, a derivation sequence $G \Rightarrow H \Rightarrow X$ via p, p' given by the following pair of double-pushouts is called sequentially independent, if there are morphisms $L' \rightarrow C$ and $R \rightarrow C'$ such that $L' \rightarrow C \rightarrow H = L' \rightarrow H$ and $R \rightarrow C' \rightarrow H = R \rightarrow H$.*



Given productions $p = (L \leftarrow K \rightarrow R)$ and $p' = (L' \leftarrow K' \rightarrow R')$ in an AHL-net transformation system the production $p + p' = (L + L' \leftarrow K + K' \rightarrow R + R')$ is called a parallel production of p and p' , provided there are binary coproducts $L + L'$, $K + K'$, and $R + R'$ that are guaranteed by the characteristics of category **AHLNET**. A direct derivation $G \Rightarrow X$ via a parallel production $p + p'$ is called a parallel derivation. The following theorem defines the relationship between parallel derivations and sequential independent productions.

Definition 13 (Parallelism Theorem) *In any HLR-system based on a HLR1-category (CAT, M) the following propositions hold:*

- **Synthesis.** *Given a sequentially independent derivation sequence $G \Rightarrow H \Rightarrow X$ via (p, p') there is a synthesis construction leading to a parallel derivation $G \Rightarrow X$ via $p + p'$.*
- **Analysis.** *Given a parallel derivation $G \Rightarrow X$ via $p + p'$ there is an analysis construction leading to two sequentially independent derivation sequences $G \Rightarrow H \Rightarrow X$ via (p, p') and $G \Rightarrow H' \Rightarrow X$ via (p', p) .*

The sequential independent derivation sequence $G \Rightarrow H \Rightarrow X$ via p and p' actually indicates that the occurrences of L in G and L' in C do not interfere with each other, in the sense that nothing is deleted that other production needs. Therefore, the sequentially independent derivations can be sequentialized in any order without affecting the final result [44]. Therefore, given a sequential independence derivation sequence $G = G_0 \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \dots \Rightarrow G_n = H$ via p_1, \dots, p_n , we may write $G \Rightarrow H$ over a production set $P = \{p_1, \dots, p_n\}$, or more specifically $G \Rightarrow H$ is a parallel derivation over P according to parallelism theorem if the corresponding coproducts exist.

2.5 Summary

As we have discussed in the previous chapter, the proposed framework integrates multiple techniques seamlessly: algebraic specifications, Petri nets, category theory, and transformation systems. This chapter gives a brief introduction for each of them as the background knowledge for the following chapters.

CHAPTER 3

COMPONENT-BASED SYSTEM MODELING

FRAMEWORK

3.1 Introduction

Currently the most popular support in industry for component-based frameworks appears to be COM+ and CORBA. Unfortunately, components in these frameworks lack a precise semantics probably due to their focus on system implementation, which makes it difficult to reason about this kind of systems. Many formal methods have been proposed to model and analyze component based systems, including Piccola Calculus [116], Abstract Behavior Types [6], and Eiffel Language [60]. In this chapter, we use Petri nets as the underlying formal method, and present a component modeling framework.

One particular concern in component-based systems is the component modeling. The generic component modeling, presented in this paper, has been mainly motivated by the ideas in [148] for “tiered component framework”, and by the concepts of “nets and rules as tokens” for Petri nets [79, 152, 153]. In [148], component frameworks are organized into multiple layers, and two layers often suffice in most cases. Blackbox frameworks accept “plug-in” components without modifications to the framework. The architecture can be extended further: a component framework itself can be slotted into a higher tier framework that regulates interactions. Such an idea is adopted in our work to separate component functionalities from message pool management and required properties such as responsiveness, scalability, security, and reliability.

More specifically, the internal behavior is captured by a function net whereas message pool management and required properties are modeled by component nets in which function nets serve as tokens.

Although components have been the predominant focus of research, they address only one aspect of component-based software development. Another important aspect is interactions among components, i.e. connectors. Connectors are sometimes deliberately modeled as components (connection components in Rapide [101]). In my work, in order to make the distinction clearer, we use a different technique – transformation rules [128] to model connectors. Although the main purpose of adopting transformation rules is to model connectors, they can also be explored to refine component nets in multiple ways to add additional functionalities such as creation and destruction of components.

This chapter introduces the proposed framework for component-based system modeling. Components' internal behaviors captured by function nets, are wrapped by component nets, which not only deal with message pool management with other components, but also model non-functional component requirements. A set of component nets are composed into a (sub)system model by applying transformation rules. Such an approach is flexible, and makes the reuse and maintenance of components and connectors easier since connectors and components are independent from each other in the framework.

This chapter is organized as follows: Section 2 outlines related works. Section 3 explains the framework informally through a dining philosopher example. The component model is described in section 4, while transformation rules are defined and classified in section 5. The integration approach and analysis techniques are illustrated in section 6 and 7, respectively. Finally a summary is given.

3.2 Related Work

Different modeling languages have been proposed to model component-based systems during last century, such as Unified Modeling Languages (UML), Cadena [66], Embedded Systems Modeling Language (ESML) [86], and Ptolemy II [28] etc.. Among them, Petri nets [112] draw attention since they are a simple, graphic based but formal modeling language, which is suitable to model concurrent and distributed systems.

The ability to compose Petri nets is fundamental to component-based system modeling. In the research literature, there are other ways to compose Petri nets to form a system model. One of them is to construct algebras of Petri nets over constants and compositional operators as in [115, 130]. In their work, labeled Petri nets are extended with interfaces (public places and transitions) through which components communicate with the external environment. Another way to compose Petri nets is through place fusion [18, 91], transition fusion [17], or both [34]. However, place fusion and transition fusion are very tightly coupled, which cannot decide the enabling of a transition locally, and even worse violate the modular principle of incremental system development. The last way to compose Petri nets is based on category theory [99] [11]. Unlike our work, there is no explicit separation among component models and their interaction models, which violates reusability and maintainability.

Among the previous works, the works of Padberg [125–127] based on category theory and Sibertin-Blanc [143, 144] based on arc fusion are the closest to ours. Padberg et al. specified a component as a model specification with an import interface IMP , an export interface EXP , and a body BOD connected by an embedding morphism $imp : IMP \rightarrow BOD$ and an substitution morphism $exp : EXP \rightarrow BOD$. IMP , EXP , and BOD are objects of Place/Transition net category. Three module operations Disjoint Union, Union, and Composition are defined to provide flat and hierarchical composition semantics for Place/Transition nets. Unlike our work,

they focused on low level Petri nets with markings as basic objects, and composition of components is always well-defined by importing and exporting functionalities, while we focus on concurrent distributed systems interacting with each other through message exchange.

Blanc [143, 144] proposed another Petri net based formalism for modeling, analysis and simulation of systems: Cooperative net and communication net, both of which can model complicated distributed systems as a set of components that have their own internal structures and behaviors, and also communicate with each other through message passing. Each component is a cooperative/communication net. Component composition is achieved through arc fusion, a looser coupling compared with place and transition fusion. Although the enabling of a transition can be judged locally, the firing of a transition is defined globally. Even worse, there is a structural dependence among components due to the potential structural reference in transition actions. More specifically, one component has to refer to other components' internal places for the purpose of communication, which is in general not available during modeling process. Therefore, structural dependence makes the reuse of components and support for incremental design harder.

Another extension of Petri nets introduces object-oriented concepts, which provides an easy understanding of modeled systems and the reusability of Petri nets. This approach is not in conflict with our work since we focus on the modeling of communication mechanisms and component interactions. More specifically, object-oriented approach can be adopted to construct function nets modeling component behavior.

3.3 Informal Introduction to the Framework

In order to illustrate concepts of the framework, we present a small system inspired by the case study “the Dining Philosophers” in [144]. In our version, philosophers,

the host, and the servant communicate with each other by sending and receiving messages.

Figure 5(a) shows our version of philosophers. A philosopher can join the table to think and eat. In order to join the table, he sends a seat request to the host. If a seat is available, the philosopher can sit in the allocated seat. When he feels hungry, he can obtain his left and right forks by asking the servant. Only with two forks in hands, he can eat. After a philosopher finishes eating, he can release forks by notifying the servant so that the servant can take back the forks. A philosopher can leave the table for reading by notifying the host.

Figure 5(b) and 5(c) show Petri nets for the host and the servant respectively. We assume there are n seats around the table, and n forks on the table. The seats are managed by the host. A philosopher can only take the seat allocated by the host. The host always let each philosopher sitting in the same seat. Forks on the table are managed by the servant. The servant can give a philosopher his left and right forks if the servant receives his request and both forks are available, i.e. no other philosophers are using them.

There are two special kinds of places in the Petri nets of Fig. 5: input places and output places. An input place represents an “unidirectional channel” through which the external environment can send messages to the model, while an output place represents a “unidirectional channel” through which the model can affect its external environment by sending message to it. In Fig. 5, an input place is denoted by a circle with a thick line, while an output place is denoted by a circle with a dashed-thick line. The set of input and output places are $\{AssignedSeat, AssignedFork\}$ and $\{RequestSeat, PhilLeft, RequestFork, ForkReleased\}$ respectively for component models of philosophers. A philosopher sends a seat request to its external environment, and the host is notified from a message at place *RequestSeat*. Whenever a message is put in place *SeatRequest*, the host knows there is a new seat request from some philosopher.

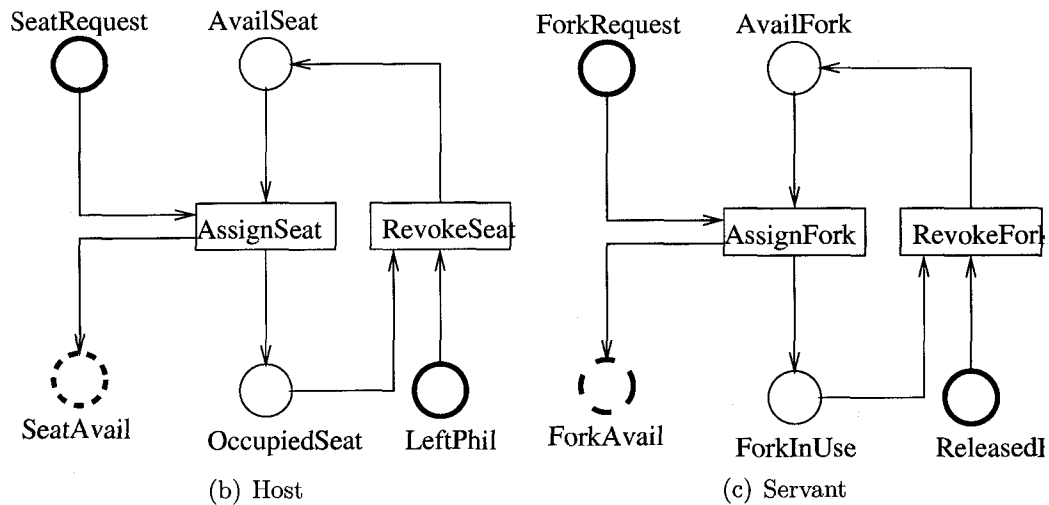
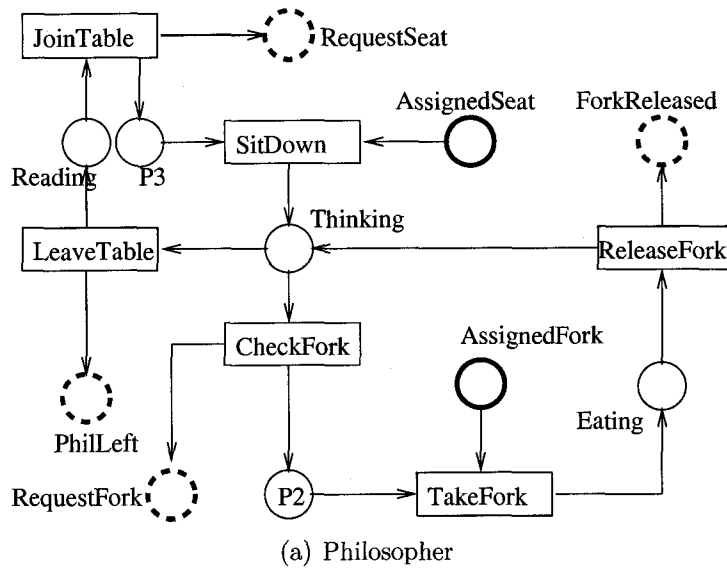


Figure 5: Component Models in Hurried Philosopher Example

Although each component in the dining philosopher problem is modeled by a Petri net, and the protocol between a component and its external environment is implicitly defined by specifying the sets of input and output places, the specification of communications among components is still missing. In other words, an approach should be proposed to integrate these individual models to a complete model without major modification. A straightforward way is through place fusion or transition fusion. In this case, an output place in one model can be merged with an input place in another model. For example, we can merge places *AssignedFork* in Fig. 5(a) with the place

ForkAvail in Fig. 5(c). However, this approach can cause several problems. First, it requires internal information of component models, which is often not needed during communication between different components. Furthermore, it breaks the principle of modularity. Second, place fusion may change the semantics of individual component model. For example, most of reactive systems respond to the next external event only when they have handled the current event just like the run-to-completion assumption in UML state machines. However, place fusion may destroy the above working order: to preserve the behavior, Petri nets have to be changed [52], which makes the synthesis more complicated. Finally it cannot distinguish channels or connectors from components based on syntax, which makes systems hard to understand.

In the framework, Petri nets in Fig. 5 are called function nets. Another kind of Petri nets called component nets is proposed to “wrap” function nets through the idea “nets as tokens” [152,153]. More specifically, function nets model component internal behavior in terms of event handling, while component nets model the management of message pools for a set of components sharing the same behavior. The object G in Fig. 6 contains component nets for philosophers and servants. Generally, a component net has the following places: a set of places called input interface receiving messages from environment, a set of places called output interface sending messages to environment, and a place P_{object} containing function nets as tokens. There is a set of input transitions, in our case only one transition t_{in} passing messages to function nets. Similarly, there is a set of output transitions, in our case only one transition t_{out} passing messages from function nets to output interface. A transition $t_{response}$ is enabled if a transition in the function net is enabled and there are no tokens in the output places. As a result of firing transition $t_{response}$, an enabled transition in the function net is fired. A component instance of a component model, described by a sub-marking of a component net, is denoted by tokens in interface places and P_{object} sharing the same identification number. For example in Fig. 6, the tokens with the

same color red in philosopher component belongs to the same component instance, i.e. there are three philosophers and one servant.

In order to construct systems based on component nets, transformation rules are adopted to specify message exchange between interfaces of different components. Here we assume that tokens contains sender and receiver information, and message parameters. Fig. 6 shows a rule and its application to component nets of philosophers and servants. A production, i.e. rule $p:L \xleftarrow{l} K \xrightarrow{r} R$ consists of three objects L, K, R and two morphisms l and r . Given a morphism $L \xrightarrow{f} G$ denoted by a dashed arrow in Fig. 6, we can apply the production to the object G (disjoint union of philosopher and servant in Fig 6). If gluing conditions are satisfied, the pushout complement X can be constructed such that the diagram (1) is a pushout. Due to the characteristics of category, object H exists such that the diagram (2) is also a pushout. Therefore, we say H is derived from G via the production p , denoted by a transformation $G \xrightarrow{p} H$. The component nets for philosophers and servants are connected through the channel denoted by R . In more general case, channel may be more complicated, such as an AHL-net with memory and buffer.

Figure 7 shows the resulted system net by synthesizing different component nets. In Fig. 7, component nets are denoted by enclosed dotted lines. We assume there is a reading philosopher *Watson*. The type of places P_i and P_o is a queue satisfying FIFO (first in, first out) in this case. Table 2 shows the firing sequence of *Watson* joining the table. In the table, *SeatNO* is the seat assigned by the host to *Watson*.

3.4 Component Models

In the proposed framework, components are modeled by component nets, a variant of Algebraic High-Level Net. Component nets explore the idea of “Nets as Tokens” proposed by Dr. Valk [152] for the introduction of object-oriented concepts into the Petri net formalism. The higher level capture the message passing between different

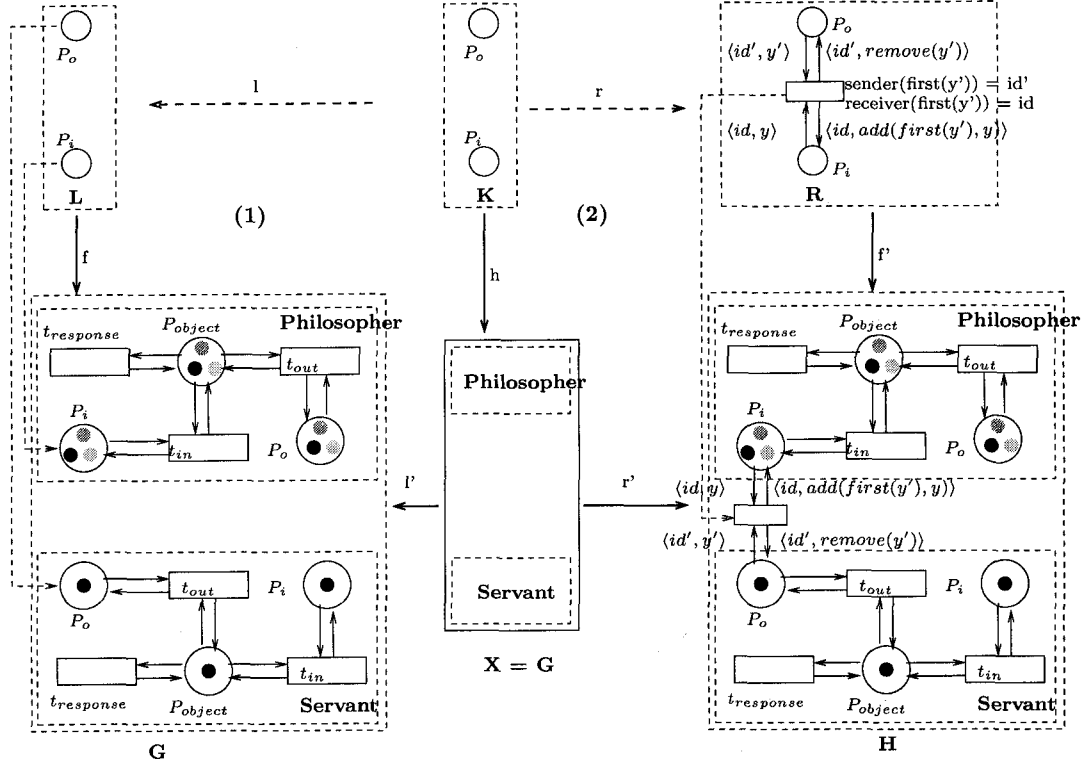


Figure 6: A Transformation Example

components, while the lower level, called function nets model component behavior. This section explain function nets and component nets in detail.

3.4.1 Function Nets

Function nets are used to model component functionalities without the concern of component interactions and non-functional requirements such as responsiveness, scalability, security, and reliability etc. More specifically, function nets specify component responses to messages from the external environment. We define function net as follows:

Definition 14 (Function Net) A *function net* is a 4-tuple $BN = (N, P_{in}, P_{out}, allocate)$ where:

- N is an AHL-net;
- $P_{in} \subseteq P$ is a set of input places;

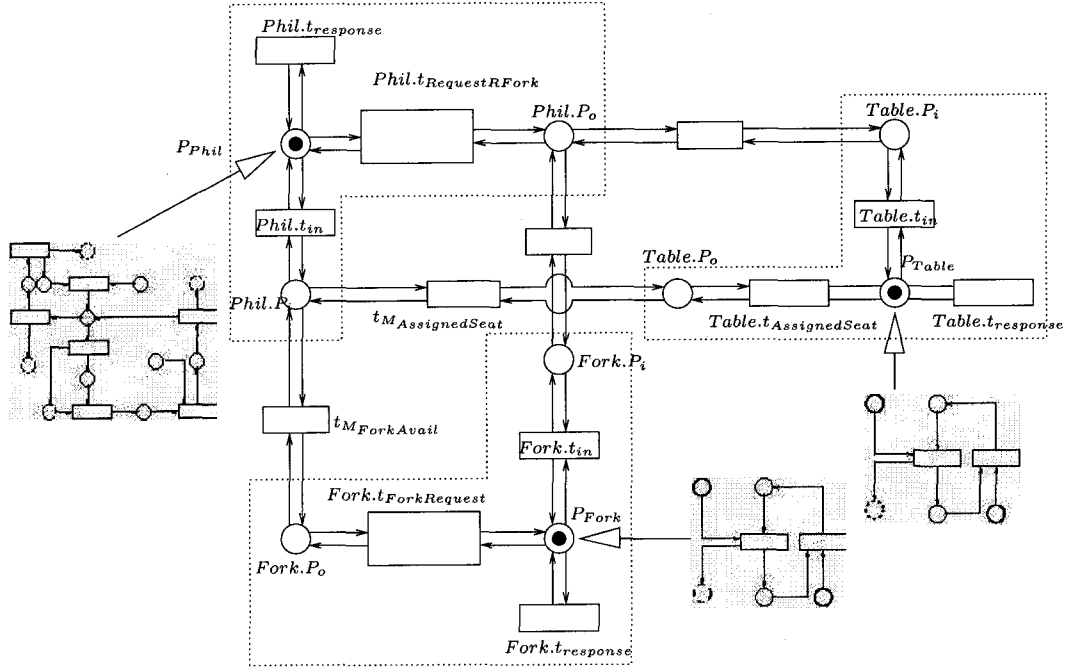


Figure 7: System Net of Dining Philosopher Example

- $P_{out} \subseteq P$ such that $\forall p \in P_{out} : p^\bullet = \emptyset$ is a set of output places disjoint from input places ($P_{in} \cap P_{out} = \emptyset$);
- *allocate* is a function assigning each input place a set of tokens it may receive from environment, i.e. $\forall p \in P_{in} : allocate(p) \subseteq 2^{A_{type(p)}}$ and $\forall p, p' \in P_{in} : allocate(p) \cap allocate(p') = \emptyset$.

A function net with a non-empty marking M of AHL-net N is stable if:

- No messages in the input places: $\forall p \in P_{in}, \nexists term \in A_{type(p)} : (term, p) \leq M$;
- No message in the output places: $\forall p \in P_{out}, \nexists term \in A_{type(p)} : (term, p) \leq M$;
- No transition is enabled under the marking M : $\forall t \in T : pre(t) \not\leq M \vee cond(t) = false$.

As the above definition indicates, function nets are a special kind of algebraic high-level nets [128] by classifying places into three categories: input places, output places

Table 2: Transition Firing Sequence of *Watson* Joining Table

	Marking of the Synthesis Net						Fired Transition	
	Phil(Watson)			Table Agent			Commun. Net	Func. Net
	P_i	P_o	Function Net	P_i	P_o	Function Net		
1	\emptyset	\emptyset	(Watson, Reading)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$	$Phil.t_{response}$	JoinTable
2	\emptyset	\emptyset	(Watson, P5) \oplus (SeatReq, RequestSeat)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$	$Phil.t_{SeatRequest}$	N/A
3	\emptyset	SeatReq	(Watson, P5)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$	$t_{M_{SeatRequest}}$	N/A
4	\emptyset	\emptyset	(Watson, P5)	SeatReq	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$	$Table.t_{in}$	N/A
5	\emptyset	\emptyset	(Watson, P5)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$ \oplus (SeatReq, SeatRequest)	$Table.t_{response}$	AssignSeat
6	\emptyset	\emptyset	(Watson, P5)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$ \ominus (SeatNO, AvailSeat) \oplus (SeatNO, OccupiedSeat) \oplus (\langle SeatAvail, SeatNO \rangle), SeatAvail)	$Table.t_{SeatAvail}$	N/A
7	\emptyset	\emptyset	(Watson, P5)	\emptyset	\langle SeatAvail SeatNO \rangle	$\Sigma_{j=1}^n(j, AvailSeat)$ \ominus (SeatNO, AvailSeat) \oplus (SeatNO, OccupiedSeat)	$t_{M_{SeatAvail}}$	N/A
8	\langle SeatAvail, SeatNO \rangle	\emptyset	(Watson, P5)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$ \ominus (SeatNO, AvailSeat) \oplus (SeatNO, OccupiedSeat)	$Phil.t_{in}$	N/A
9	\emptyset	\emptyset	(Watson, P5) \oplus (\langle SeatAvail, SeatNO \rangle), SeatAvail)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$ \ominus (SeatNO, AvailSeat) \oplus (SeatNO, OccupiedSeat)	$Phil.t_{response}$	SitDown
10	\emptyset	\emptyset	(\langle Watson, SeatNO \rangle , Thinking)	\emptyset	\emptyset	$\Sigma_{j=1}^n(j, AvailSeat)$ \ominus (SeatNO, AvailSeat) \oplus (SeatNO, OccupiedSeat)		

and internal places. Input places contain messages received from the external environment. Output places contain messages to the external environment as responses, while internal places indicate component status. Input, output, and internal places are supposed to be disjoint, otherwise the meaning of a message in such places is ambiguous. Upon reception of a message in an input place, a function net can be executed until reaching a stable status, in which no transition is enabled and the component is waiting for the next to-be-handled message. As a result of handling a received message, in most cases, at least one message in an output place is generated as a response to the external environment.

In most cases, sets of input and output places are not empty. However, if both sets are empty, we say such a component is a closed system that does not interact with other components or systems. If only the set of input places is empty, we say the component is a message generator, which affects its environment. If only the set of output places is empty, the component is called recorder, which only records environment's influence on itself without feedback.

Given a component, we can either model its behavior as a function net from scratch, or make little modifications to the available Petri net behavioral model to meet the definition of function nets. However, it is in general impossible to model or obtain the behavioral model of commercial-off-the-shelf (COTS) components. What we have known about these blackbox components is the well-defined relationship among interfaces (input and output places). Fortunately, we can either construct a behavioral model from such relationships or use algebraic specifications to represent such interface relationships of blackbox components. In either way, component nets work correctly since a function net is actually treated as an algebraic specification due to the fact that Petri nets are monoids [108].

3.4.2 Component Nets

A component not only has its own behavior, which is modeled by function nets, but also needs to communicate with other components through message exchange, and may have some non-functional requirements including responsiveness, scalability, security, and reliability. Therefore, we adopt the idea “nets as tokens” to synthesize function nets with communication mechanism. The paradigm “nets as tokens” was introduced by Valk in order to allow nets as tokens, called object nets, within a net, called system net [152, 153]. The object nets may not only change its marking, but also modify its net structure in the context of system nets. Such characteristics, together with the communication complexity between objects nets and system nets, confine the research of object nets on low level Petri nets.

Fortunately, net structures of function nets in our work are supposed to be unchangeable. Therefore, an algebraic high level net may be represented by an algebra, which can be adopted by another algebraic high level net as part of its specification, which is viable since Petri nets are monoids [108]. The following shows the signature SIG_{BN} constructed for a given function net $BN = (N, P_{in}, P_{out}, allocate)$ where $N = (SPEC, X, P, T, type, cond, pre, post, A)$ and X a finite set of variables, i.e. $X = \{x_1, \dots, x_n\}$.

$SIG_{BN} =$

sorts: Transitions, Places, InPlace, OutPlace, Bool, System, InEvent, OutEvent,
 Domain $_{x_1}$, ..., Domain $_{x_n}$

opns: truthValue, falseValue: \rightarrow Bool

enabled: System \times Transition \times Domain $_{x_1}$ \times ... \times Domain $_{x_n}$ \rightarrow Bool

enabled': System \rightarrow Bool

fire: System \times Transition \times Domain $_{x_1}$ \times ... \times Domain $_{x_n}$ \rightarrow System

hasoutput: System \times OutPlace \times Events \rightarrow Bool

hasoutput': System \rightarrow Bool

hasinput: System \times InPlace \rightarrow Bool

hasinput': System \rightarrow Bool

output: System \times OutputPlace \times OutEvent \rightarrow System

input: System \times InEvent \rightarrow System

Operation *enabled* specifies if a transition is enabled under the current marking and the assignment to variables. Operation *fire* fires a given transition with a given variable assignment. Operation *hasoutput* checks if a given output place contains a given message. Operation *hasinput* checks if a given input place contains a message. Operations *enabled'*, *hasoutput'* and *hasinput'* are the more abstract version of corresponding operations. Operation *output* removes a given message from a given output

place, while operation *input* adds a given message to an input place. Based on the signature SIG_{BN} , a SIG_{BN} -algebra B can be constructed as shown in Appendix A.

In order to communicate with the environment, each component manages an input and output message pool. Although we choose the data structure queue in this investigation, the message pool actually can be represented by any other data structures such as list or stack. However, no matter what kind of abstract data structure is chosen, following signature SIG_{com} should be “included” in specifications of message pools.

```

 $SIG_{com} =$ 
  sorts: Queue
  import: MESSAGE
  opns: empty:  $\rightarrow$  Queue
        add: Queue  $\times$  Message  $\rightarrow$  Queue
        remove: Queue  $\rightarrow$  System
        first: Queue  $\rightarrow$  Message

```

where

```

 $MESSAGE =$ 
  sorts: Message
  import: NAME, ID
  opns: kind: Message  $\rightarrow$  Name
        sender: Message  $\rightarrow$  ID
        receiver: Message  $\rightarrow$  ID

```

Operation *add* adds a message to the queue, while operation *remove* removes first available message from the queue. Operation *first* returns the first available message in the queue. Operations *kind*, *sender*, *receiver* return message type, message sender and receiver respectively. The signature *NAME* and *ID* specify the message type and object unique identification number respectively.

Based on the above algebraic specifications, we can define component nets as the following:

Definition 15 (Component Net) *Given a function net BN , a **component net** based on BN is an AHL-net $N_0 = (SPEC_0, X, P_0, T_0, type_0, cond_0, pre_0, post_0, A_0)$ shown as L in Fig. 9 where*

- $SPEC_0 = (SIG_{BN} + SIG_{com}, \emptyset)$ is an algebraic specification;
- $P_0 = \{P_{object}, P_i, P_o\}$;
- $T_0 = \{t_{out}, t_{response}, t_{in}\}$;
- $type_0(P_{object}) = ID \times B_{System}$, $type_0(P_i) = ID \times Queue_{in}$, $type_0(P_o) = ID \times Queue_{out}$;
- *Function $cond_0$ is as follows:*
 $cond_0(t_{in}) = cond_0(t_{out}) = \emptyset$;
 $cond_0(t_{response}) = (\exists t \in B_{Transition}, \exists v_{xi} \in A_{si} \text{ such that } xi \in X_{si}, \text{ for } i=1, \dots, n:$
 $enabled_B(x, t, v_{x1}, \dots, v_{xn}) == true)$;
- *The function pre_0 is as follows:*
 $pre_0(t_{in}) = (\langle id, y \rangle, P_i) \oplus (\langle id, x \rangle, P_{object})$;
 $pre_0(t_{response}) = (\langle id, x \rangle, P_{object})$;
 $pre_0(t_{out}) = (\langle id, y \rangle, P_o) \oplus (\langle id, x \rangle, P_{object})$;
- *The function $post_0$ is as follows:*
 $post_0(t_{in}) = (\langle id, remove(y) \rangle, P_i) \oplus (\langle id, input_B(x, first(y)) \rangle, P_{object})$;
 $post_0(t_{response}) = (\langle id, fire_B(x, t, v_{x1}, \dots, v_{xn}) \rangle, P_{object})$;
 $post_0(t_{out}) = (\langle id, add(e, y) \rangle, P_o) \oplus (\langle id, output_B(x, p, e) \rangle, P_{object})$;
- A_0 is a $SPEC_0$ -algebra.

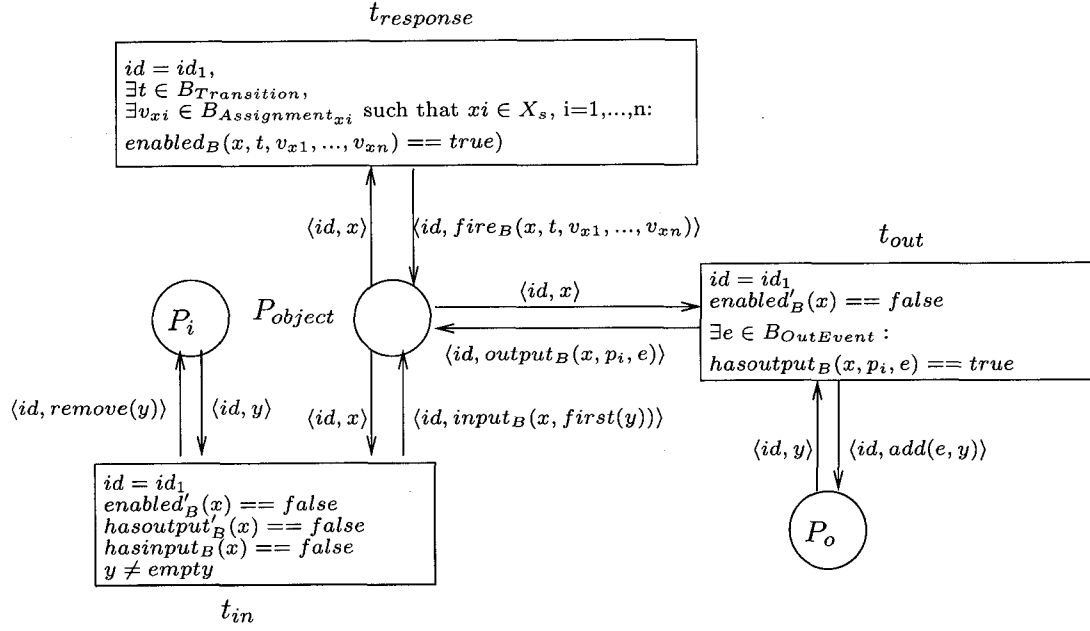


Figure 8: The Semantics of a Component

A component net not only executes its function net as a response to the external environment, but also manages the input and output message queues according to system specification. Generally these tasks are not isolated from each other, rather there is a strong relationship between them affecting component behavior in terms of:

- When to fetch from the input queue the next message, which is ready to be processed by the function net?
- When to put a generated message to the output queue in which the message is available to its environment?
- When to process the current message in input places by executing the function net?

A component specifies the answers to the above questions for all of its components. The AHL-net N_0 as constructed in Definition 15 is a special component architecture that allows its components to execute the function net and manage the queues at

appropriate time. Due to the flexibility of potential productions, a component may provide complex answers to the above questions for contained components. In other words multiple components of a component may have different semantics if necessary. For example, in Fig. 8 a component id_1 (id_1 is a constant of sort ID) is distinguished from other components by having run-to-completion assumption: A component can handle the next message in the input queue only if the function net is stable, while it can put a generated message to the output queue when no transition in the function net is enabled. Therefore, refinement productions are introduced to provide flexibility to model complex answers to the above questions by refining transitions t_{in} , t_{out} , $t_{response}$ and places P_i and P_o .

3.5 Transformation Rules

Besides modeling components, we need to provide an approach to model interactions in the form of message exchange as well as a methodology to integrate component models into a system model in a modular and incremental way.

In the framework, transformation rules (or productions) in HLR-category ($AHLNET$, M_{AHLN}) [128] are adopted to model interactions. By exploring transformation rules, the framework has the following advantages as well as flexibility and powerful expressiveness:

- Transformation rules have formal semantics. Since Petri nets are also a formal graphic modeling language, our methodology of system modeling has a strong theory basis.
- By adopting transformation rules, we not only separate component modeling from channel/connector modeling, but also distinguish dynamic component creation and destruction from component modeling.
- Transformation rules can also be explored to refine/construct component nets.

- By adopting transformation rules, system can be modeled in a modular and incremental way.
- It is flexible to model different (sub)systems containing various aspects or scenario by applying different transformation rules to different component nets.

I have defined multiple types of transformation rules for various purposes in the framework. Table 3 gives a summary of production types. Creation/destruction message passing productions are distinguished from interaction productions since such messages are passed from a component to a component net, not from a component to another component just like interaction productions. Refinement productions are used to refine component nets, especially the relationship between message pools and function nets to support complicated behavior such as run-to-completion assumption in UML state machine diagrams.

We have to point out that currently we do not take message broadcasting into account. Given a transformation rule $(L \leftarrow K \rightarrow R)$ and an AHL-net G , the occurrence of L in G is not unique, and therefore we may get multiple AHL-nets. Not all of them are valid (sub)systems with the concern of requirements. A consistent condition is given for each kind of productions to guarantee that the system model we obtain is valid. In the rest of this section, I give the definition of each kind of transformation rules.

3.5.1 Refinement Rules

Definition 16 (Refinement Production) *Given a component net N_0 based on the function net BN , a **refinement production** $p: (L \leftarrow K \rightarrow R)$ is in the form of Fig. 9 such that $L = N_0$ and morphisms $K \rightarrow L$ and $K \rightarrow R$ are in the class M_{AHLN} . Additionally, for any transition in R with an incoming arc from place P_{object} , there is an corresponding outgoing arc to place P_{object} .*

In Fig. 9, a dashed rectangle in R indicates a sub-AHL-net, the structure of which is up to each production. Therefore, a refinement production actually specifies that

Types	Description
Refinement Production	Refining generic component nets
Creation Production	Adding dynamic component creation functionality to component architectures
Destruction Production	Adding dynamic component destruction functionality to component architectures
Interaction Production	Connecting component architectures through message exchange
Message Creation Passing Production	Passing creation messages from a component to a component net
Message Destruction Passing Production	Passing destruction messages from a component to a component net

Table 3: Summary of Production Types

place P_i may be refined with places P_i^1, \dots, P_i^n , place P_o with places P_o^1, \dots, P_o^m , transitions $t_{in}, t_{response}, t_{out}$ with sub-AHL nets. The set of places P_i^1, \dots, P_i^n is called the input interface of CA , similarly P_o^1, \dots, P_o^m is the output interface. In refinement productions, firing a transition in R either updates the marking of a concrete function net, or never need access to tokens in place P_{object} . Such restriction is for the purpose of property “uplifting” specified in Section 3.7

3.5.2 Creation Rules

During system evolution, component instances¹ may be created and destroyed dynamically during runtime, which has to be supported by system designs. In order to support dynamic instantiation of components, the following productions are introduced.

Definition 17 (Creation Production) *A creation production $p: (L \leftarrow K \rightarrow R)$ is in the form of Fig. 10 such that:*

- *Morphisms $K \rightarrow L$ and $K \rightarrow R$ are in the class M_{AHLN} ;*
- *The morphism $K \rightarrow L$ is an isomorphism;*

¹In general components are heavyweight units with exactly one instance in a system. However our approach can also be applied to model systems made up of objects. Therefore, the term – component instance– is a little bit abused.

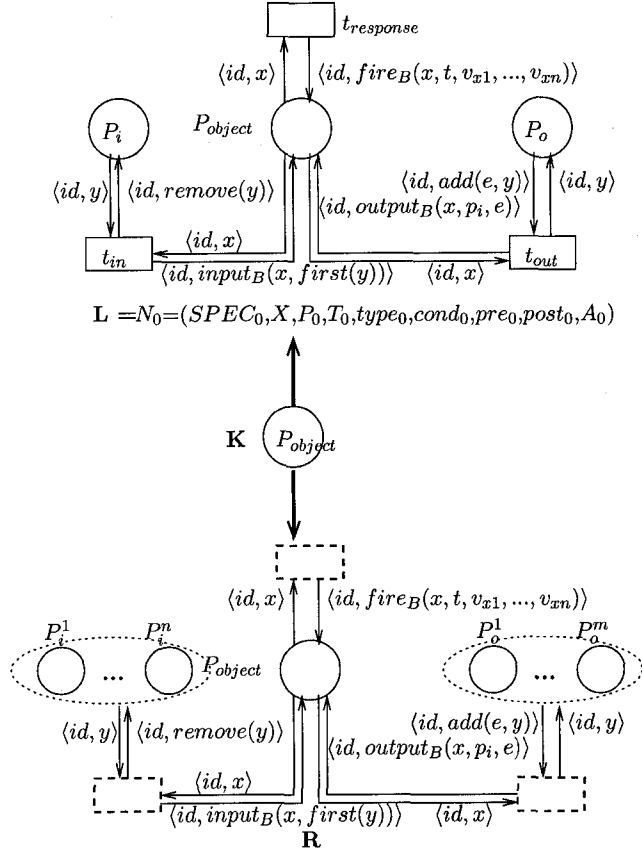


Figure 9: A Refinement Production

- R is an AHL-net, and the dashed rectangle represents a sub-AHL-net specified by each production.
- The output tokens along arcs from the dashed transition should have the same identification number.

In the Fig. 10, place P_c contains creation request, while place P_{id} indicates next available unique identification number that will be assigned to next constructed component. The dashed rectangle represents a sub-AHL-net specifying the process of creation request. Similarly, we can define destruction productions.

3.5.3 Destruction Rules

Definition 18 (Destruction Production) A *destruction production* $p: (L \leftarrow K \rightarrow R)$ is in the form of Fig. 11 such that:

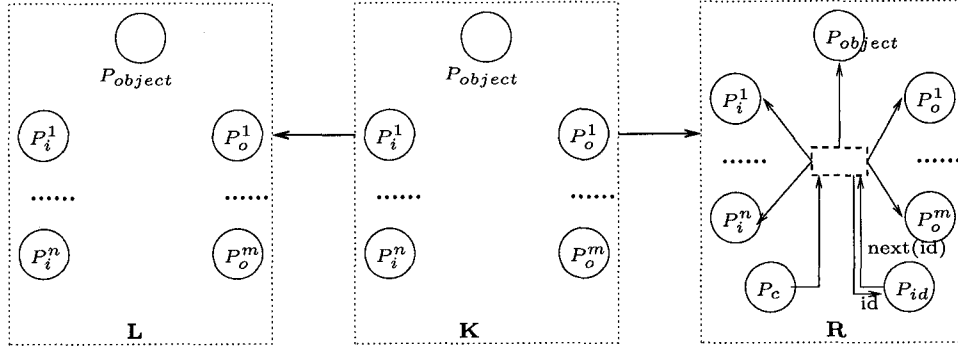


Figure 10: A Creation Production

- Morphisms $K \rightarrow L$ and $K \rightarrow R$ are in the class M_{AHLN} ;
- The morphism $K \rightarrow L$ is an isomorphism;
- R is an AHL-net, and the dashed rectangle represents a sub-AHL-nets specified by each production.
- The output tokens along arcs to the dashed transition should have the same identification number.

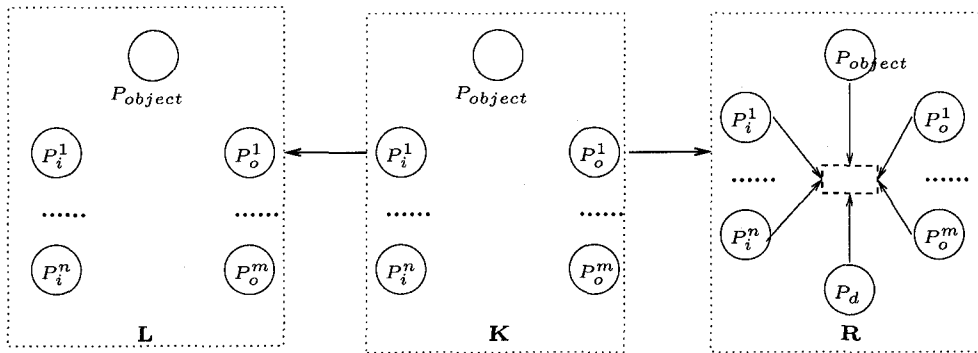


Figure 11: A Destruction Production

By applying refinement productions and creation/destruction productions to a component net as constructed in Definition 15, we can obtain a more refined AHL-net, called component, which is the atomic entity in a component-based system. A component specifies potential communication mechanisms, behaviors, and their relationship

for a set of components. Consistent transformation is introduced to constrain the applications of the above productions.

Definition 19 (Consistent Transformation) *Given a production $p:L \xleftarrow{l} K \xrightarrow{r} R$, a transformation $G \xrightarrow{p} H$ where the occurrence of L in G is $f = (f_{SPEC}, f_P, f_T, f_A):L \rightarrow G$, is a consistent transformation if the following conditions hold:*

- *When p is a refinement production:*
 $f \in M_{AHLN}$ and morphisms f_P and f_T are isomorphisms.
- *When p is a creation/destruction production:*
 $f \in M_{AHLN}$, $\{f_P(P_i^1), \dots, f_P(P_i^n)\}$ and $\{f_P(P_o^1), \dots, f_P(P_o^m)\}$ are the input and output interface of G respectively.

A derivation sequence $G_0 \xrightarrow{p_1} G_1 \xrightarrow{p_2} \dots G_{n-1} \xrightarrow{p_n} G_n$ is called a consistent derivation sequence if $G_i \xrightarrow{p_{i+1}} G_{i+1}$ is a consistent transformation for $i=0, \dots, n-1$.

Definition 20 (Component Architecture) *Let N_0 be a component net over the function net BN as constructed in definition 20. Given an AHL-net transformation system $ATS = (N_0, \mathbb{P})$, a **component architecture** CA is an AHL-net such that there is a refinement production p , a creation production p' and a destruction production p'' in \mathbb{P} satisfying one of the following consistent derivation sequences:*

- $CA = N_0$;
- $N_0 \xrightarrow{p} CA$;
- $N_0 \xrightarrow{p} CA' \xrightarrow{p'} CA$;
- $N_0 \xrightarrow{p} CA' \xrightarrow{p''} CA$;
- $N_0 \xrightarrow{p} CA' \xrightarrow{p'} CA'' \xrightarrow{p''} CA$;
- $N_0 \xrightarrow{p} CA' \xrightarrow{p''} CA'' \xrightarrow{p'} CA$;

A marking M of a component architecture CA is a well-defined marking if for any identification number $id \in ID$: $(\langle id, \langle BN_{id}, M \rangle \rangle, P_{object}) \leq M \Leftrightarrow (\langle id, in \rangle, P_i^k) \leq M \Leftrightarrow (\langle id, out \rangle, P_o^l) \leq M$ for $k = 1, \dots, n$ and $l = 1, \dots, m$. If $(\langle id, \langle BN_{id}, M \rangle \rangle, P_{object}) \oplus \bigoplus_{k=1, \dots, n} (\langle id, in \rangle, P_i^k) \oplus \bigoplus_{l=1, \dots, m} (\langle id, out \rangle, P_o^l) \leq M$, we say there is a component instance id of the component architecture. The marking $(\langle id, \langle BN_{id}, M \rangle \rangle, P_{object}) \oplus \bigoplus_{k=1, \dots, n} (\langle id, in \rangle, P_i^k) \oplus \bigoplus_{l=1, \dots, m} (\langle id, out \rangle, P_o^l)$ is called the snapshot of the component instance id .

3.5.4 Interaction Rules

Component architectures describe a set of components sharing the same function net structures but with different queue structures and behaviors. However, there is limited benefits without providing an approach to integrate them into a single model, which supports modular and incremental design. Two components interact with each other by exchanging messages, which is modeled as productions.

Definition 21 (Interaction Production) An *interaction production* $p: (L \leftarrow K \rightarrow R)$ is in the form of Fig. 12 where

- Morphisms $K \rightarrow L$ and $K \rightarrow R$ are in the class M_{AHLN} ;
- The morphism $K \rightarrow L$ is an isomorphism;
- R is an AHL-net such that: $sender(first(y')) = id'$ and $receiver(first(y')) = id$;

An interaction production actually models an unidirectional communication channel between two component architectures. By replacing id and id' with concrete components, the production models the unidirectional communication channel between two components of different component architectures. A channel modeled by an AHL-net has its own properties and characteristics. It can be a pipeline, a unreliable network, or a FIFO structure, and it may have its own message buffer. Therefore

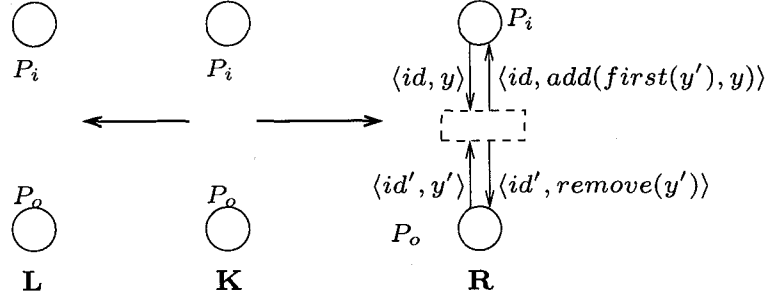


Figure 12: An Interaction Production

using productions to model interactions between components provides the flexibility to handle different situations by separating concerns in the process of system modeling. We have to point out that places P_i and P_o may be mapped into the same component (architecture) in the occurrence mapping. In this case, it models the communication between the same component (architecture).

3.5.5 Creation/Destruction Message Passing Rules

The messages of creation and destruction are distinguished from other messages that are sent from one component to another component (Currently, we do not consider message broadcasting.) since the receiver of such a message is not a component, but a component architecture. The ultimate reason is due to the fact of object-oriented concepts that it is class, not an object to create or destroy an object. Therefore, we have to introduce new productions to transfer creation or destruction messages from a component to a component architecture.

Definition 22 (Creation/Destruction Message Passing Production) *Given a component architecture CA , a **creation message passing production** $p: (L \leftarrow K \rightarrow R)$, describing creation message passing to CA , is in the form of Fig. 13 where*

- *Morphisms $K \rightarrow L$ and $K \rightarrow R$ are in the class M_{AHLN} ;*
- *The morphism $K \rightarrow L$ is an isomorphism;*

- R is an AHL-net satisfying: $\text{kind}(\text{first}(y)) = \text{Creation}$, $\text{sender}(\text{first}(y)) = \text{id}$, and $\text{receiver}(\text{first}(y)) = \text{CA}$. In the figure, the dashed rectangle represents a sub-AHL-net specified by each production.

Similarly, by replacing place P_c with place P_d we can describe the destruction message passing to component architecture CA .



Figure 13: A Creation Message Passing Production

The following theorem summaries the relationship among interaction productions and creation/destruction message passing productions.

Theorem 1 *Let N_i be component architectures over function nets BN_i for $i=1, \dots, n$; and \mathbb{P} a set of interaction productions and creation/destruction message passing productions. Given any two productions $p, p' \in \mathbb{P}$, the derivation sequence $G = N_0 + \dots + N_n \Rightarrow H \Rightarrow X$ via p and p' is sequentially dependent.*

It is easy to prove the above theorem since any two sequential independent productions do not delete any part of original AHL-net.

Definition 23 (Valid Transformation) *Given a production $p=L \xleftarrow{l} K \xrightarrow{r} R$, a transformation $G \xRightarrow{p} H$ where the occurrence of L in G is $f = (f_{\text{SPEC}}, f_P, f_T, f_A):L \rightarrow G$, is a valid transformation if the following conditions hold:*

- When p is an interaction production:

$f \in M_{AHLN}$ and $f_P(P_i)$ belong to the input interface of some component architecture, and $f_P(P_o)$ belong to the output interface of some component architecture.

- When p is a creation/destruction message passing production:

$f \in M_{AHLN}$, and $f_P(P_o)$ belong to the output interface of some component architecture, and $f_P(P_c)$ is the creation place of some component architecture.

A derivation sequence $G_0 \xrightarrow{p_1} G_1 \xrightarrow{p_2} \dots G_{n-1} \xrightarrow{p_n} G_n$ is called valid derivation sequence if $G_i \xrightarrow{p_{i+1}} G_{i+1}$ is a valid transformation for $i=0, \dots, n-1$.

3.6 Component Composition

Definition 24 (System) Let N_i be component architectures over function net BN_i for $i=1, \dots, n$; and $ATS = (N_1 + \dots + N_n, \mathbb{P})$ an AHL-net transformation system where \mathbb{P} is a set of interaction productions and creation/destruction message passing productions. A **system** (SYS, M) is an AHL-net with well-defined marking such that: $\exists P \subseteq \mathbb{P}$ such that $N_1 + \dots + N_n \Rightarrow SYS$ is a valid derivation sequence over P ;

According to the parallelism theorem and the above theorem, the AHL-net SYS exists and does not depend on the order of application of rules in P . The components in the system (SYS, M) is decided by the well-defined marking M , i.e. the projection of M over each component architecture is a well-defined marking.

3.7 Analysis

We now analyze function nets and systems defined in Definition 24. More specifically, an approach is proposed to check if a Petri net is a function net. Additionally, we show the way to model checking the system net derived from the framework.

3.7.1 Function nets

Not all algebraic high-level nets can serve as function nets. A function net has a finite behavior given an initial marking. Additionally, a component should be capable of handling any messages put in one of the input place when it is in a stable snapshot.

Definition 25 (Function Net Property) *A function net must satisfy the following properties:*

- 1 *A function net cannot have an infinite firing sequence from any marking M such that $\langle N, M \rangle = \text{input}_B(\langle N, M_0 \rangle, e)$ where M_0 is a stable marking and $e \in Q_m$.*
- 2 *For any stable marking M , given an event e , there is a place $p \in \mathcal{R}(e)$ such that there exists an enabled transition $t \in P^\bullet$ under the marking $M \oplus (e, p)$.*

These two properties ensure that a component eventually will respond to all messages in its input queue. It is easy to check that Petri nets in Fig. 5 are not function nets because they violate the second property. For example, a thinking philosopher cannot handle a fork available message although this message is not correct with regard to the status of the philosopher. In other words, those function nets cannot handle unexpected messages, which in general indicate a design error. In order to make them function nets, exception handling transitions and exception recording places are added as dashed rectangles and circles respectively in Fig. 14. By introducing exception recording places, it is easier to check the occurrence of unexpected messages during model checking.

A function net is an open system since the message sequences it handles is variable and decided by its environment in runtime. Such a characteristic makes it hard to check whether a Petri net is a function net. In other words, we cannot provide a general rule or theorem to judge if a Petri net satisfies the above two properties. However, we can make sure a Petri net satisfying the above two properties if they meet some conditions, though vice versa is not always correct.

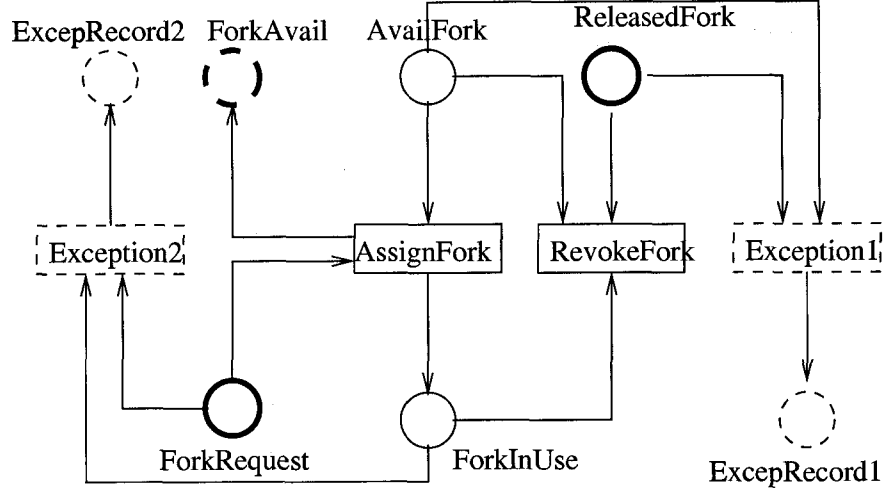


Figure 14: The Valid Function Net of the Servant

Theorem 2 *An AHL-net satisfies property 1 if for any loop $p_1, t_1, p_2, t_2, \dots, t_{n-1}, p_n = p_1$ where $p_i \in \bullet t_i \wedge p_{i+1} \in t_i^\bullet$ $i=1, \dots, n-1$, there is an input place $p \in P_{in}$ not in the loop such that $\exists k : p \in \bullet t_k \wedge p = \emptyset$.*

The proof is straightforward. Actually, theorem 2 means that any potential infinitely firing sequence needs “assistance” from its input queue. An incoming place cannot be in a loop since it has no incoming arcs.

Theorem 3 *An AHL-net satisfies property 2 if for any $p \in P_{in}$, there is a set of transitions $\{t_1, \dots, t_n\} \subseteq p^\bullet$ where $\bullet t_i = \{p\}$ such that for any assignment ass to variables $X: \forall_{i=1}^n ass(cond(t_i)) = true$*

3.7.2 System nets

System nets derived from the framework according to the Definition 24, unlike function nets, are closed Petri nets, which means a variety of traditional Petri net analysis techniques can be applied to detect errors and check the correctness of a model with regard to some properties specified in requirements. In this paper, we focus on exploring model checking technique to check if a synthesis net is correct

with regard to specified LTL (Linear Temporal Logic) formulae. There are some off-the-shelf model checking tools such as SPIN [80], SMV [105]. However, we choose Maude [48] as our analysis tool due to the characteristics of Maude.

Unlike other model checking systems, Maude is a high-performance reflective language supporting both equational and rewriting logic specifications and rewriting logic computation, which makes Maude applicable to many potential application areas – beyond traditional ones such as hardware and communication protocols. For example, the potential application areas are hard to specify for SPIN, which is designed and optimized for distributed algorithm applications, because SPIN enforces the communication between processes through FIFO channels and has limited support for data types. In addition to the more expressive power, Maude has a collection of formal tools supporting different forms of logic reasoning to verify program properties, including [36]:

- a model checker to verify LTL properties of finite-state system modules;
- an inductive theorem prover to verify properties of functional modules;
- a Church-Rosser checker, to check such a property for functional modules;
- a Knuth-Bendix completion tool and termination checker for functional modules; and
- a coherence checker for system modules.

Specific to algebraic high-level nets, it has several advantages to take Maude as the model checking tool. First, its specification language is an enhanced form of algebraic specifications, which makes transformation from ALHN to Maude easier and automated. Second, the Maude LTL model checker can model check systems whose states involve data in data types of infinite cardinality, which is crucial for model checking high level Petri nets. Third, in addition to model checking, we can use the inductive theorem prover directly without major modification to the specification.

Furthermore, the Maude LTL model checker is comparable to other high-performance model checker such as SPIN in time and space performance [48].

It is straightforward to transform SIG_{BN} to a functional modules in Maude that defines data types and operations on them by means of equational theories. Functional modules also support multiple sorts, subsort relations, operator overloading, and assertions of membership in a sort. The functional module for SIG_{BN} -Algebra for component architecture *servant* in dining philosophers problem is shown in Appendix B. The Petri net simulation can be defined by a system module in Maude. A system module specifies a rewrite theory, which has sorts, kinds, operators, and can have three types of statements: equations, memberships, and rules, all of which can be conditional. The system module for component *servant* is also shown in the Appendix B. In our implementation, a rule is defined for each transition with a valid assignment. For example, there are two rules *Tresponse_AssignFork* and *Tresponse_RevokeFork* for the transition $t_{response}$, each of them corresponds to a valid assignment.

Generally, when we talk about LTL property of a plain Petri Net model, the atomic predicate is in the form of $p(a)$, which is satisfied by the Petri net if place p contains a token a under current marking M , denoted by $M \models p(a)$. However, it is inconvenient to express properties of synthesis nets using such atomic predicates since a token itself can be a Petri net. Therefore, a new kind of atomic predicates is introduced for AHL-nets of component-based system.

Definition 26 (Predicates and Formulae of Petri Nets)

- For any simple type A_s , we assume that there is a set of propositional formulae Φ_s , each of which specifies a subset S_φ of A_s . A predicate φ under an element e of A_s is valid if:

$$e \models_{A_s} \varphi \iff e \in S_\varphi$$

- For each product type $A_{s_1} \times A_{s_2}$, the associated set of formulae is defined as $\Phi_{s_1} \times \Phi_{s_2}$. A formula (φ_1, φ_2) under an element (e_1, e_2) is valid if

$$(e_1, e_2) \models_{A_{s_1} \times A_{s_2}} (\varphi_1, \varphi_2) \iff e_1 \models_{A_{s_1}} \varphi_1 \wedge e_2 \models_{A_{s_2}} \varphi_2$$

- Let $N = (SPEC, X, P, T, type, cond, pre, post, A)$ be an AHL-net and M is a marking of N . The set of predicates of AHL-net N and its semantics are defined as the following:

- For each place p and a formula φ of the type $type(p)$, $p(\varphi)$ is a predicate. A predicate $p(\varphi)$ is valid if:

$$\langle N, M \rangle \models_N p(\varphi) \iff \exists a \in A_{type(p)} : a \models_{A_{type(p)}} \varphi \wedge (a, p) \leq M$$

- A formula of AHL-net N is constructed by boolean connectors \neg, \wedge, \vee , and additional connectors $\bar{\wedge}$ such that

$$\begin{aligned} \langle N, M \rangle \models_N \neg p(\varphi) &\iff \forall e \in A_{type(p)} : e \not\models_{A_{type(p)}} \varphi \\ \langle N, M \rangle \models_N (p_1(\varphi_1) \wedge p_2(\varphi_2)) &\iff (\langle N, M \rangle \models_N p_1(\varphi_1)) \wedge \\ &\quad (\langle N, M \rangle \models_N p_2(\varphi_2)) \\ \langle N, M \rangle \models_N (p_1(\varphi_1) \vee p_2(\varphi_2)) &\iff (\langle N, M \rangle \models_N p_1(\varphi_1)) \vee \\ &\quad (\langle N, M \rangle \models_N p_2(\varphi_2)) \\ \langle N, M \rangle \models_N (p_1(\varphi_1) \bar{\wedge} p_2(\varphi_2)) &\iff \exists e_1 \in A_{type(p_1)}, e_2 \in A_{type(p_2)} : \\ &\quad e_1 \models_{A_{type(p_1)}} \varphi_1 \wedge e_2 \models_{A_{type(p_2)}} \varphi_2 \wedge \\ &\quad (e_1, p_1) \oplus (e_2, p_2) \leq M \end{aligned}$$

The connectors \wedge and $\bar{\wedge}$ are equivalent if $p_1 = p_2$. Otherwise, there is a small difference, i.e. $p(\varphi_1) \bar{\wedge} p(\varphi_2) \Rightarrow p(\varphi_1) \wedge p(\varphi_2)$ since $p(\varphi_1) \bar{\wedge} p(\varphi_2)$ describes the case

that there are two tokens in place p satisfying predicates φ_1 and φ_2 respectively, while there can be only one token in place p satisfying $\varphi_1 \wedge \varphi_2$. The predicate $Fork.P_{object}(AvailFork(fork1))$ describes markings of the servant such that place P_{object} contain a token $\langle N, M' \rangle$ where the number 1 fork is available under the marking M' .

In order to model checking system nets obtained through the approach proposed in previous sections, we need to “uplift” properties of lower level Petri nets to upper level, i.e. from function nets to component nets since it is awkward to model checking properties of a token. Such idea is not viable in general since lower level Petri nets may appear in different places. Fortunately, in a system net, the token representing a concrete function net is always in the same place P_{Object} . Therefore, a formula of a function net id always has an counterpart in the system net. This relationship is defined by the following definition, which only considers future time operators \square , \diamond and \mathbf{U} .

Definition 27 *Function \mathcal{M} maps a future time LTL formula φ of a function net N of component instance id to a future time LTL formula of the system net in the following way:*

- If φ is a propositional formula:

$$\mathcal{M}(\varphi) = P_{Object}((id, \varphi))$$

where (id, φ) is a formula of the product type $ID \times System$ defined in Section 3.4.

- If $\varphi = ?\varphi'$ where $?$ is a future time operator \square , or \diamond , and φ' is a formula of function net N :

$$\mathcal{M}(\varphi) = ?\mathcal{M}(\varphi')$$

- If $\varphi = \varphi' \mathbf{U} \varphi''$ where φ' and φ'' are formulae of function net N :

$$\mathcal{M}(\varphi) = \mathcal{M}(\varphi') \mathbf{U} \mathcal{M}(\varphi'')$$

Theorem 4 *Let N_c be a component net as a part of a closed system net, and N_f be a function net of a component instance. Let φ be a LTL formula of N_f . Then φ is satisfied by N_f if and only if the formula $\mathcal{M}(\varphi)$ is satisfied by the system net.*

The proof is straightforward due to two facts: First, A function net stays in the same place during its lifetime. Second, the marking of a function net is also a part of global marking of system nets. Some atomic predicates defined in the component architecture philosopher is shown in the Appendix B. The module *SIG_BN_PHIL_PREDS* defines atomic predicates for function net of philosopher, while the module *PHIL_PREDS* defines atomic predicates for component architecture of philosopher. In the module *PHIL_PREDS*, the formulae of function nets is expressed as a part of condition instead of parameters of formulae of component nets.

We have checked mutual exclusion and starvation properties of dining philosopher problem. Mutual exclusion property means two adjacent philosophers cannot eat at the same time:

$$\square((pPSPOBJ\text{-}Eating(phil1, phil1, 1) \wedge pPSPOBJ\text{-}Eating(phil2, phil2, 2)))$$

Starvation property means if a philosopher wants to eat, he will eventually get a chance to eat:

$$\square(pPSPOBJ\text{-}P2(phil1, phil1, 1) \rightarrow \diamond(pPSPOBJ\text{-}Eating(phil1, phil1, 1)))$$

Unfortunately, starvation property does not hold, i.e. a philosopher may starve to death although he has sent his request to the servant. The counterexample shows

that the problem is because the system always responds to other philosophers' request such as leaving table and joining table. Therefore, the philosopher who wants to eat is "stuck" after he sent his fork request, and never got a chance to obtain response from the servant. The starvation property should be hold with fairness constraint. Unfortunately, the Maude LTL model checker does not support the fairness constraint.

3.8 Summary

A framework to model component-based system in an incremental way is proposed in this chapter. The framework separates concerns of component models and their interaction models explicitly. A component is modeled by an algebraic high-level Petri net. By introducing the idea of "net as tokens" to algebraic high-level Petri nets, we can model more complex components due to the flexibility in handling the relationship between component behavior and communication mechanism. Component interactions are specified by productions based on HLR-category (**AHLNET**, M_{AHLN}). Additionally, productions can also be explored to refine component behavior and its relationship with communication mechanism, and model functionality of dynamic component creation and destruction. In the framework, different techniques are synthesized seamlessly.

In order to analyze system nets constructed through the framework, model checking is explored to verify component properties. Model checking is very effective and verification is completely automatic. We have used Maude in the running example. The translation is straightforward. Although we translated the example to Maude function and system modules manually in this case, the translation process can be fulfilled automatically since each firing of a transition can be viewed as a rewriting step of current marking. However, model checking has its own limitation – it is in general not applicable to infinite state systems. Specific to Maude, it cannot handle complicated system nets in terms of net structure and involved sorts since searching next applicable rewriting rule is time and space consuming.

There are two kinds of properties to be verified: component behavior property and communication protocol property. Verification of component behavior property generally involves one component, while verification of communication protocol property involves several even the whole system nets, which may be quite large in some situations. To solve this problem, we are investigating several compositional model checking techniques. Among the various proposed automated compositional verification techniques in temporal logic [19,35,64] and in Petri nets [85,154], we found that the interface module technique [19] and the IO graph technique [154] are most relevant to our research. We are currently focusing on how to adapt these compositional verification technique to analyze system nets obtained through our framework. We are also studying compositional temporal logic proving techniques developed in [1].

CHAPTER 4

VERIFICATION AND VALIDATION OF UML DESIGNS

4.1 Introduction

Unified Modeling Language (UML) [120], the de facto object-oriented modeling language, supports multi-view approach, i.e. artifacts created in the development process for different system aspects are modeled and analyzed in various kinds of UML concepts. More specifically, class diagrams specify system static structure; statechart diagrams describe behavior of individual classifiers; activity diagrams emphasize control flows and object flows for coordinating low-layer behaviors, rather than which classifier own those behaviors; interaction diagrams including sequence diagrams and communication diagrams illustrate implementation of use cases by describing interactions among objects to complete tasks.

The multi-view and multi-notation approach helps designers focus on individual viewpoints so that models are more manageable and less error-prone. However, inconsistencies arise because “the models overlap – that is they incorporate elements which refer to common aspects of the system under development – and make assertions about these aspects which are not jointly satisfiable as they stand, or under certain conditions” [147]. The detection of inconsistencies is not easy due to the multi-notations. Generally speaking, there are four broad approaches to detect inconsistencies in software models: the logic-based approach, the model checking approach, the specialized model analysis approach and the human-centered collaborative exploration [147]. In the UML community, most of the research explore the third approach

to detect inconsistencies of UML models, i.e. UML models are translated into a common semantic domain. UML inconsistency detection is even more difficult since syntax and semantics of UML are informal and imprecise compared to formal specification languages. Although UML inconsistency has been widely studied, a majority of them focus on the formalization of individual diagrams and only check consistency within one or between two diagrams.

In this chapter, the framework proposed in Chapter 3 was explored to detect UML inconsistency among multiple diagrams, more specifically class diagrams, statechart diagrams, activity diagrams, interaction diagrams including sequence diagrams and communication diagrams. Component nets are constructed from class diagrams, activity diagrams and statechart diagrams, while transformation rules are extracted from interaction diagrams. A (sub)system net can be acquired by applying a set of transformation rules to a set of component nets. Various kinds of UML inconsistencies can be detected by exploring different analysis techniques on derived (sub)system nets.

The rest of the chapter is organized as the following: section 2 provides an overview of related works on UML. The formalization of UML diagrams to obtain two-layer AHL-nets and transformation rules are specified in section 3. The inconsistencies based on Petri nets are defined and detected in section 4. Finally, a summary is given.

4.2 Related Works

This section introduces the related works on the formalization of UML diagrams and UML inconsistency detection.

4.2.1 Formalization of UML Diagrams

UML, as a family of languages, lacks precise semantics since static and dynamic semantics of UML diagrams are defined in plain English language, which is inherited

ambiguous. Therefore, lots of formal languages have been adopted to provide precise semantics for various UML diagrams for the purpose of analysis.

Abstract State Machines

Abstract State Machines (ASM) [82], proposed more than 10 years ago, were initially used to provide operational semantics for programming languages. Later, due to its ability to simulate any algorithm without implementing them, it was explored for high level design and analysis. In past several years, ASM was used to provide a formal and more precise semantics for UML.

There are two approaches to formalize UML based on ASM. One is to formalize UML diagrams on meta-model level [118]. The UML meta-model is a subset of class diagrams. All other diagrams including class diagrams are defined by the meta-model. Therefore, the formalization of UML meta-model gives precise semantics for all other diagrams. However, this makes it hard to analyze UML models based on the semantics. The other approach is to formalize UML diagrams such as activity diagrams [23], statechart diagrams [24,37], class diagrams and object diagrams [142].

Graph Transformation

Graph transformation [43], also known as graph rewriting or graph reduction, combines advantages of graphs and rules into a single computation paradigm. “It has been studied in a variety of approaches, motivated by application domains such as pattern recognition, semantics of programming languages, compiler description, implementation of functional programming languages, specification of database systems, specification of abstract data types, specification of distributed system etc” [4]. Since UML itself is a diagrammatic language, it seems reasonable and promising to apply techniques developed in the graph transformation field to UML.

At first, graph transformation was applied to classic *Statecharts* in [103]. Later, different UML diagrams were formalized by graph transformation, such as class diagrams in [58], statechart diagrams in [56,57,92,104], collaboration diagrams in [50,76],

and sequence diagrams in [67]. However, these works only focused on one diagram, which make them impossible to detect inconsistency between diagrams. Therefore, here we only discuss the work of [59] and [93].

Using graph transformation, [93] and [59] propose an approach to integrate class diagrams, object diagrams, statechart diagrams, sequence diagrams and collaboration. More specifically, they defined a system state as an object diagram that is extended with object states and event queues. Then graph transformation rules can be derived from class diagrams and statechart diagrams. The graph transformation rules associated with class diagrams defines semantics for each operation in class diagrams, while graph transformation rules associated with statechart diagrams define the semantics for each transition. By combining these two kinds of graph transformation rules, the change of system states as a response to events can be defined. Collaboration diagrams and sequence diagrams can be verified based on system states with these rules.

Their work is different from my research. First, they still did not formalize class diagrams, although they view the semantics of class diagrams as the set of valid object diagrams. Therefore, we cannot detect inconsistencies related with class diagrams. Second, graph transformation is a variant of term rewriting. Although they can be executed or explored to prove some properties, generally speaking, analysis based on them is hard and few tools support their analysis. Finally, we cannot have a clear idea about semantics of each operation in class diagrams until detail design. Thus, the advantage of their work cannot be explored at early stage.

Prototype Verification System

The Prototype Verification System (PVS) [124] is a formalism for design and analysis of system specifications. The PVS environment consists of a PVS specification language [123] based on classical, typed higher-order logic, an interactive theorem prover [140] and other tools.

It has been shown that UML diagrams can be formalized by PVS. For example, class diagrams are formalized in [7, 10], statechart diagrams in [8, 149] and sequence diagrams in [9]. However these formalizations are separated from each other. Therefore, only single diagrams can be analyzed based on this method, which is not enough for our research goal. Additionally, PVS specification language is based on high order logic, which is not well suited to model dynamic behavior.

Object-Z

Object-Z [145] is an object-oriented extension of the Z formal specification language. During last several years, Object-Z was used to formalize UML diagrams, such as class diagrams in [87, 89], statechart diagrams in [88, 90], and collaboration diagrams in [5]. However, no efforts have been made to integrate them into a complete Object-Z schema. Also due to the property of Object-Z, it is not suitable to specify the dynamic behavior, and there is few techniques and tools to support the analysis of Object-Z or Z specifications.

Algebraic Specification and LOTOS

Algebraic specification [45, 46] was used to formalize UML class diagrams [3, 27, 53]. However, algebraic specification is best at the description of abstract data type, it is hard and inconvenient to model the system behavior by itself. Therefore, algebraic specification has to be combined with other formal languages to model systems. The Language of Temporal Ordering Specification (LOTOS) [25] and Enhanced-LOTOS [150], which combine algebraic specification (ACT-ONE [45]) and algebraic processes such as Communication Sequential Processes (CSP) [78] and Calculus of Communicating Systems (CCS) [110], was explored to formalize UML diagrams, such as class diagrams in [38], statechart diagrams in [38, 77, 159]. In these works, only [38] considered the transformation from class diagrams and statechart diagrams to LOTOS. However, the connection between LOTOS theories derived from both diagrams was ignored. Additionally collaboration/sequence diagrams was still not formalized.

Another problem of LOTOS is that LOTOS is hard to read and calculus of algebraic process is not powerful enough to model the full dynamic behavior of UML.

Petri Nets

Petri nets, as a graphic modeling language for concurrent and distributed systems, have a close relationship with UML – State machine diagrams have a similar semantics to Petri nets and activity diagrams are defined in Petri net semantics. Additionally, Petri nets can be used as a complement to UML [29,84] during software development as well as the semantic domain to formalize UML diagrams [15,40,41,72–74,81,136]. Unlike previous works on the formalization of UML diagrams using Petri nets, we use Petri nets to construct a complete behavioral model for each class from multiple diagrams instead of an individual diagram.

4.2.2 Inconsistency Detection

Inconsistency among multiple goals, requirements or models is a active research topic in software engineering. In the 90s,, it was concluded that it is not necessary to maintain absolute consistencies among software development because by doing so, it hinders the concurrency during software development and limit the design freedom [54]. In many cases, it may be desirable to tolerate or even encourage inconsistency, “to facilitate distributed collaborative working, to prevent premature commitment to design decisions, to ensure all stakeholder views are taken into account” [117].

During last 10 years, lot of helpful results were obtained. For example, new logics such as paraconsistent logic [31,42,146], Quasi-Classical logic [83], and techniques such as the derivation of “boundary conditions” by goal regression [157] and the detection of inconsistency using pattern of divergences [156] were developed to detect and reason about inconsistency. Additionally, different solutions to inconsistency were proposed such as tolerating inconsistency [14] and “Lazy” consistency [114]. However, most of the research was focused on requirement engineering. In other words, the research of inconsistency on other phrases of software development process is ignored.

UML, the de facto object-oriented modeling language, covers all the stages of software development process, not only the requirements capture, but also system design and detail design. Therefore, UML inconsistency detection is a new challenge for software engineering community. Even worse, although UML is a modeling language, it actually consists of multiple diagrams with their own notions and terms. During two recent workshops on UML inconsistency [94, 95], lot of concrete inconsistencies were discussed and different techniques were proposed to detect specific inconsistencies. However, there are no systematic work to detect UML inconsistencies. Currently, only a simple classification of UML inconsistency (vertical v.s. horizontal, inter v.s. intra, syntax v.s. semantics) were recognized by the community.

There are two ways to analyze UML diagrams: model checking and theorem proving. Model checking is used to check if a given predicate is satisfied against the model by exploring all of its possible execution pathes. Currently, we can translate UML diagrams, especially statechart diagrams and collaboration diagrams, directly to input languages of model checkers such as PROMELA or SPIN [80]. The research in [109, 139] take this way. While another way is to formalize UML diagrams based on a semantic domain, and then the model of the semantic domain is translated into input languages of model checker.

In our research, the latter approach is adopted since model checking is not the ultimate purpose of our research. Our purpose is to analyze UML diagrams based on a semantic domain. Model checking is just one of the analysis method we take. However, there are some benefits we can obtain by combining these two approaches of model checking UML diagrams. For example, it is impossible to prove the correctness of formalization of UML diagrams directly. But by checking the same properties against the same UML diagram through these two approaches, we can increase our confidence about the formalization.

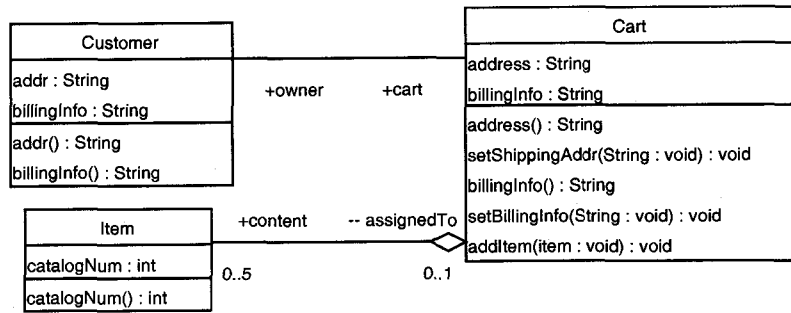
4.3 Running Example

The running example is a simple online shopping system as shown in Fig. 15. There are only three classes: *Customer*, *Cart* and *Item*. Since the owner of the store thinks their price is so low, each customer can only buy no more than five items each time.

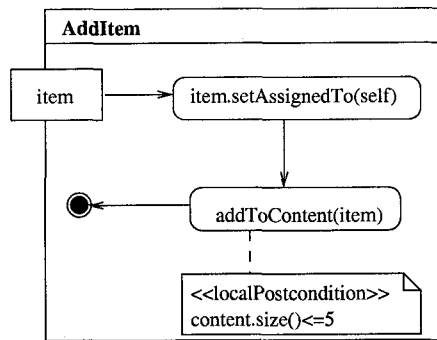
The Fig. 16 shows the UML formalization of the online shopping system. From the figure we can see classes in class diagrams consist of operations and attributes (The relation association is treated as an attribute of associated classes). The class attributes and constructors are described as algebraic class specifications, while operations defined in activity diagrams are formalized as a Petri net. Class behaviors are specified by statechart diagrams, which are formalized as Petri nets. The execution of an activity in statechart diagrams is represented as a transition in Petri nets, which is later refined by a Petri net derived from an associated activity diagram. By refining all executions of activities, we now obtain function nets for all classes. Based on function nets, component nets are constructed with regard to policies of event pools of class instances. Transformation rules are extracted from interaction diagrams based on messages passed between instances of multiple classes. An AHL-system, as we discussed in the previous section, can be constructed based on derived Petri nets and transformation rules. It is flexible to derive Petri nets models of simple systems describing single scenarios or complex systems containing all scenarios described in UML diagrams.

4.4 Algebraic View of UML Class Diagrams

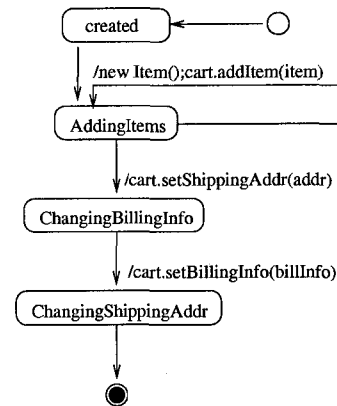
In this section we sketch the main transformation rules for UML concepts of class diagrams into algebraic specifications. We assume that for each attribute there is one or more operations that only read or update the attribute value. Such operations are called primitive operations. The access to attributes is through the invocation of these primitive operations. The classes with primitive operations and constructors can be



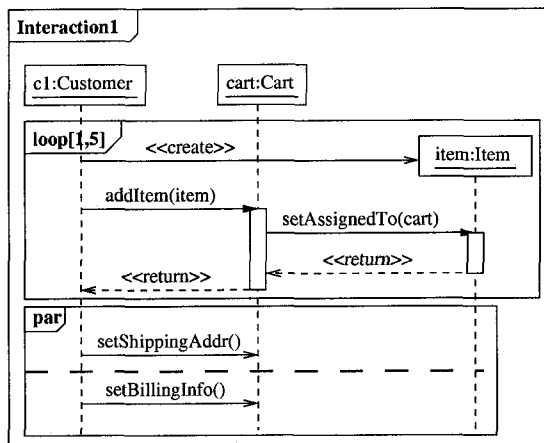
(a) Class Diagram



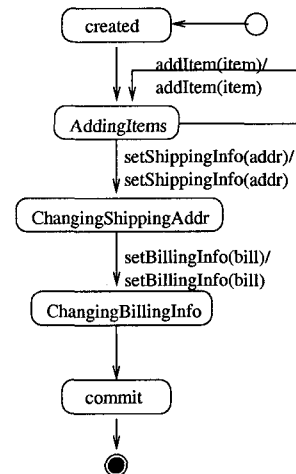
(b) Activity Diagram



(c) Customer



(d) Sequence Diagram



(e) Cart

Figure 15: A Simple Online Shopping System

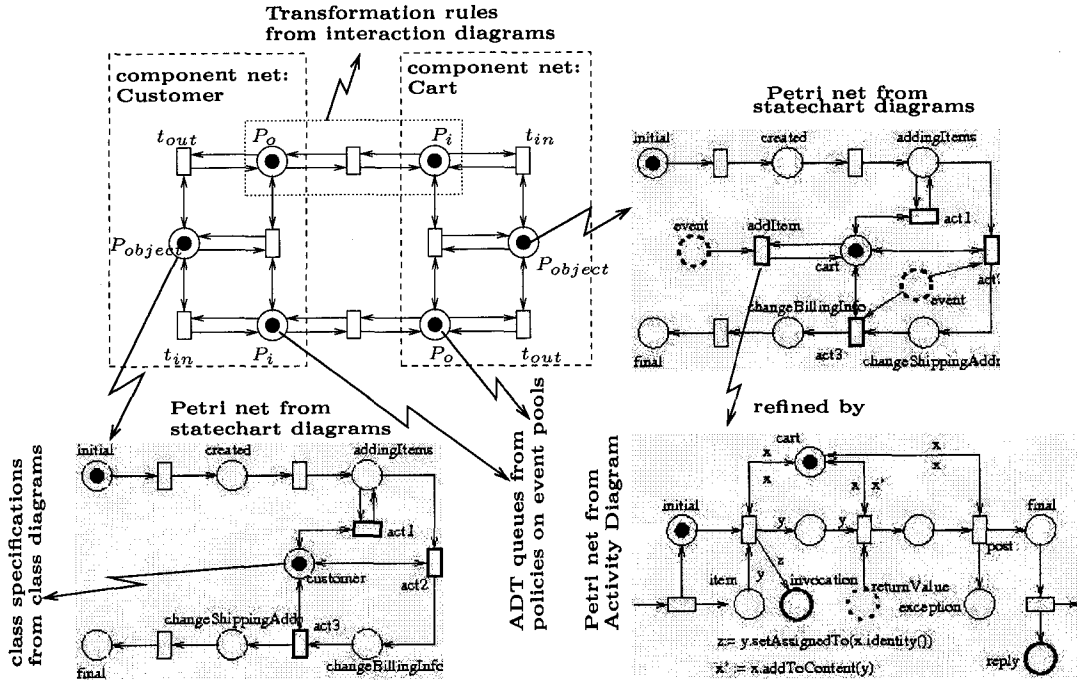


Figure 16: UML Formalization Pattern

formalized by abstract data types (ADTs). An instance of a class is represented by a value of the ADT associated to that class. Each ADT value is assigned an identity, which is treated as an explicit read-only attribute.

Among different approaches to describe main features of object-oriented concepts, we adopt the work of [129] to specify classes since it is more close to the class notations in UML. In particular, a class specification C_{spec} consists of five algebraic specifications: PAR (parameter part), EXP_i (instance interface), EXP_c (class interface), IMP (import interface) and BOD (implementation part), and five algebraic specification morphisms: $PAR \rightarrow IMP$, $PAR \rightarrow EXP_i$, $EXP_i \rightarrow EXP_c$, $IMP \rightarrow BOD$, and $EXP_c \rightarrow BOD$.

A class maps onto an ADT with constructors, primitive operations and constraints. The constructors express the instantiation process. Each ADT at least has one constructor. If no constructor is explicitly declared in class diagrams, a default public constructor without arguments is specified in the class specification. Normally, the

instance attributes are the parameters of the constructors. Some constraints are introduced for constructors to specify attribute values of created instances. The above generated operations are distributed to the instance interface, class interface or implementation according to the visibility of primitive operations and constructors.

An association declares that there can be links between instances of the associated types. A link is a tuple with value for each end of the association, where each value is an instance of the type of the end. A navigable end is an attribute ([120], Page 80), therefore, a binary association can be treated as an attribute of classes of association ends. For an association with $N > 2$ ends or an AssociationClass is formalized as a class with properties. This class maintains the set of links among association ends. Like the binary associations, a link refers to values of association ends through their identities instead of their value. By doing so, we can isolate association structures from structures of related classes. Therefore, it is possible to predefine association classes, which share almost the same instance and class interface, but have different implementations and constraints. There are two special associations: aggregation and composition. We treat aggregations as plain association, therefore no special action is needed. Composition is a form of special aggregation with strong ownership such that the part is created by the whole and the whole destroys the part before itself is destroyed. However, the order and the way in which part instances are created or destroyed is a semantic variation point, and generally not specified in class diagrams. In our work, we assume the part instances are created and destroyed as parts of the creation and destruction of whole instances, which is modeled in creation and destruction transformation rules

4.5 Formalization of State Machine

State machines are generally used to express the behavior of part of a system. In our work, its usage is more specific: describing the behavior of classes declared in class diagrams. Petri nets are similar to flat state machines in terms of states/places

and transitions connecting them. Variant Petri nets have been used to provide a more formal semantics [20, 40, 81, 107, 137]. Our previous work [40] is adopted as the foundation (Please refer to Appendix C for the formalization of UML state machine). Although it is based on hierarchical predicate/transition nets, the approach itself can be easily applied to obtain AHL-nets. However, several modifications to the original method are proposed to meet the definition of two-layer AHL-nets and therefore provide a better understanding of state machines in Petri net concepts. More specifically, the most important principle of the modification is to use component nets to model state machines by separating concerns of behavior from concerns of policies on event pools that are a semantic variant in the UML white book [120]. Therefore, event pool and run-to-completion assumption are modeled by the upper-layer Petri nets, while the lower-layer Petri nets (i.e. function nets) only specify the responses of state machines to events. By doing so, function nets exactly model the behavior of statechart diagrams, nothing more and nothing less.

The following summarizes the modification to the approach of formalizing state machines based on Petri nets in [40]:

- The place *INPUT* is replaced by a set of places, each of which is served as an input place for a distinct type of events.
- The place *OUTPUT* is replaced by a set of places, each of which is served as an output place for a distinct type of events.
- An additional place serving as the holder for the class sort specified in previous section is added by connecting all transitions that need to access the instance's attributes or methods.
- Each activity is represented by a transition, which is later refined by a Petri net derived from the corresponding activity diagram (see section 4.6).

- For each synchronous operation call or signal, a message with return value is replied to the sender, otherwise the return value (if exists) is ignored.
- A component net is constructed based on function nets obtained from previous steps with the consideration of the run-to-completion assumption (Fig. 8 and the policies of event pools. The policies on event pools of instances primarily describing the order of dequeuing and the size of the pool are specified by users in addition to UML diagrams.

The first two modifications is to meet the definition of function nets, while the third modification is to integrate class specifications with its behavior. However, we have to point out that the resulted Petri nets are not complete since activities are not supported by class specifications and need to be refined later as the above fourth item shows. Only after the integration with Petri nets derived from activity diagrams (discussed in the next section), resulted Petri nets for statechart diagrams are complete in terms of syntax and semantics.

4.6 Formalization of Activity Diagrams

Activity diagrams represent UML activity graph expressing sequence, choices and parallel execution of actions. Activities may describe procedural computation, in this context class operations, which is the only usage of activity diagrams in our work. More specifically we only consider following actions in activity diagrams: *InvocationAction* (including *CallOperationAction*, *SendSignalAction*, and *SendObjectAction*), *ReplyAction*, *CreateObjectAction*, *DestroyObjectAction*, *AcceptEventAction*. Since associations are explicitly treated as classes, the link related operations are treated as normal invocation actions.

In UML 2.0, “activities are redesigned to use a Petri-like semantics instead of state machines” [120]. Although one author advised that this statement is “only a metaphor for flow modeling without implying a complete mapping to Petri nets” [22], the metaphor can be made concrete to provide a better understanding of its

semantics. The available works of transforming activity diagrams to Petri nets [16, 22, 39, 96] emphasize data and control flow of actions, but the semantics of actions themselves is missing. We try to overcome this problem by connecting the actions with corresponding class specifications. In our work, we restrict to intermediate activities. We do not consider exceptions and structure features such as activity group and swimlane of activity diagrams. In our work, instead of passing objects, only object ID is passed in the actions.

An action is generally represented by a Petri net, as showed in Fig. 17. An asynchronous *CallOperationAction* with pre- and post-conditions is formalized in the way that pre-condition is first tested, then an event for the invocation of the operation is sent to the target, finally the post-condition is tested. Pre- and post-conditions are explicitly formalized as conditions of corresponding transitions in Petri nets. For a synchronous *CallOperationAction*, an additional place is added to receive return value so that the activity can continue. The dashed place containing the specified event in Fig. 17(c) is an input place in the derived Petri net from the corresponding state machine. An object node is represented by a place containing the object ID. The initial and final nodes are also formalized as a single place. Each parameter of activity diagrams (if exists) is described as a place. The transformation of other control nodes such as joint, fork and choices are similar to the work in [16]. The Fig. 16 shows a Petri net derived from the activity diagram *addItem*.

The transition t representing an activity in Petri nets PN_s from statechart diagrams can be refined by a Petri net PN_a derived from associated activity diagrams in following steps:

- Delete the transition t and related arcs from the Petri net PN_s ;
- Add the Petri net PN_a to the Petri net PN_s ;
- Add a new transition t_b such that its incoming places are the incoming places

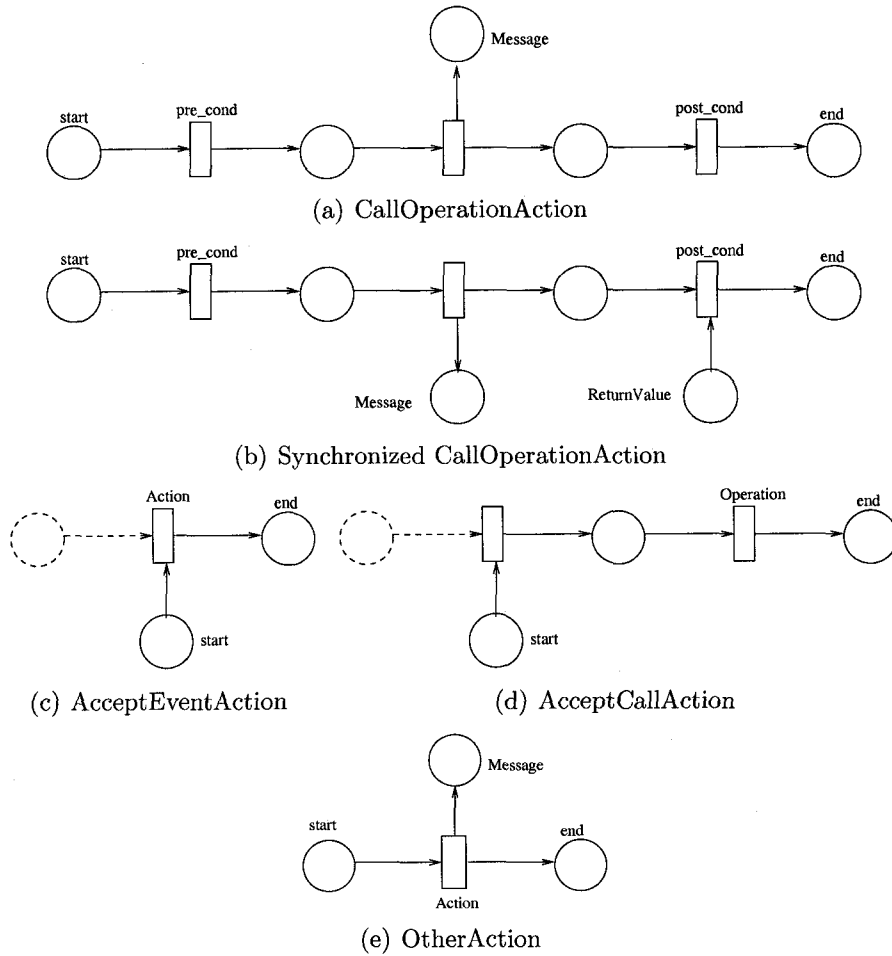


Figure 17: Petri Net Representation of Actions

of the transition t in PN_s and its outgoing places are the places in PN_a corresponding to parameter nodes and the initial node. The responsibility of the transition is to extract parameters of the activity from event parameters and start the activity.

- Add a new transition such that its incoming place is the place in PN_a corresponding to the final node, and its outgoing places is the outgoing places of transition t in PN_s . The firing of the transition indicates the end of the activity.

4.7 Transformation Rules From Interaction Diagrams

In the previous sections, we discussed the approach to obtain a component net for each class from related class diagrams, statechart diagrams and activity diagrams. In order to achieve system modeling based on Petri nets, transformation rules are necessary to integrate these components into a system. In this section, we explain how to obtain transformation rules from interaction diagrams, more specifically sequence diagrams and communication diagrams.

In our framework, transformation rules are used to model the communication/channel between objects, which is happen to be the concept of messages in interaction diagrams. For each message, we can identify sender, receiver, and message type; and therefore a transformation rule can be constructed. Fig. 18 shows the transformation rule corresponding to the message *addItem* from the *customer* to the *cart* showed in Fig. 15(d). The transition t_{pass} can be replaced by a Petri net that models more complicated channel for the message passing. A message has a property to indicate if it is a synchronous call operation or a synchronous signal, which expects return value before the sender can continue. Asynchronous message do not expect a reply message. The statechart diagram of the receiver is responsible to distinguish synchronous messages from asynchronous messages and response with a reply message to the sender.

The creation and destroy messages should be handled different since an object cannot create or destroy itself. Such messages should not be handled by instances, but by classes, component net in our case. The transformation rules for creation or destroy messages are similar to Fig. 18 except the place P_i , which is replaced by the place $P_c(P_d)$. The Place $P_c(P_d)$ is added to the component net after application of the corresponding creation (destroy) production, which also models the creation (destroy) of part instances if a composition association exists.

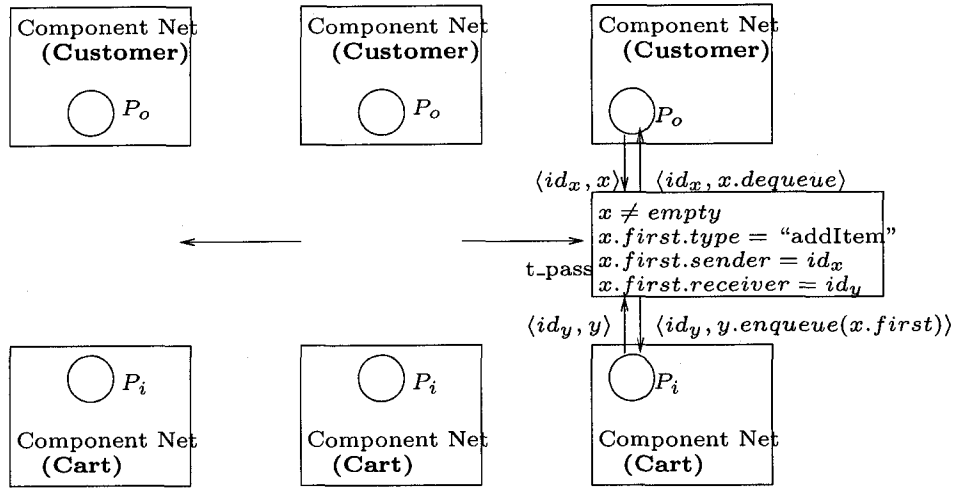


Figure 18: Transformation Rule for Passing Message *addItem*

4.8 Model Inconsistency

There is no standard definition of consistency. Multiple approaches have been proposed to define consistency according to different purposes. Consistency can be defined based on logic such that false information can be derived from multiple viewpoints [146, 146], while it can also be defined as the existence of a physical model which implements multiple viewpoints of a system model. In our work, we view inconsistency as properties or rules that system nets must satisfy. The consistency problem of an individual diagram has been widely studied in the UML community. Some of their work such as analysis of class diagrams based on algebra [3] can be adopted directly without difficulty. Therefore, in this paper we focus on the inter-consistency between different viewpoints of a system model.

First, the syntactic inconsistencies between different diagrams are represented as syntactic errors of derived component nets. There is no way to guarantee the syntactic correctness of component nets and function nets because errors do exist due to the inconsistency between UML diagrams. For example, in statechart diagrams, a variable in guards of transitions is neither an event parameter, an attribute of a class nor a role name of an association that the class of the statechart diagram can access

directly or indirectly, then in the corresponding function net, there is a transition whose condition refers to an undefined variable. In this example, we detect that the attribute *billInfo* referred to by the statechart diagram of class *Customer* does not exist. It actually should be *billingInfo*. Such kinds of inconsistencies are easy to detect, even in UML itself. Therefore, in the rest of this chapter we assume all derived Petri nets according to the proposed approach in the previous section are correct in syntax.

Most of UML consistencies can be specified as safety properties that Petri nets must satisfy during its lifetime. The safety properties are generally specified as linear temporal logic formulae, in which a kind of predicates is introduced in the form of $P(t)$, which is true if place p contains a token t under the current marking M . Therefore, a Petri net with the initial marking M_0 satisfies a safety property φ if each reachable marking from M_0 satisfies φ . Violation of a safety property implies the occurrence of an inconsistency. This kind of inconsistencies can be detected through model checking. Several safety properties for a component net of a class are showed in the following:

- The state machine diagram of a class should response to all messages/events sent to it in activity diagrams or interaction diagrams. In particular class *Cart*, such property is expressed as:

$$\square(\forall id \in ID, x \in Cart.B_{System} : cart.Object(id, x) \wedge x \neq undef)$$

since the function net becomes undefined whenever an unexpected message is input according to the SIG_{AHLN} -algebra B (see the Section 3.4). This property is violated in this online shopping example since the class *Cart* cannot handle the operation call message *setShippingAddress* from *Customer*. The original operation the customer wants to invoke is actually *setShippingAddr* instead of *setShippingAddress*.

- Visibility checking – Other objects can only access its public roles, attributes and operations. Such property for class *Item* is expressed as:

$$\begin{aligned} \square(\forall q \in Queue : Item.P_i(q) \wedge first(q).sender == first(q).receiver \wedge \\ first(q).type == OperationCall \wedge \\ first(q).OperationName == "setAssignedTo") \end{aligned}$$

This means that only instances of *Item* can invoke the method *setAssignedTo* since the corresponding association is a private one as showed in the Fig. 15(a). This property is violated due to an invocation of this method from class *Cart*, described in the activity diagram *AddItem*. By carefully reviewing these models, it is better to change the visibility of *assignedTo* to public.

- Incomplete interaction – The interaction is not complete if a message is “stuck” in the output queue of an instance in the execution of associated Petri nets. This can occur if an interaction misses some link. This can be represented by a safety property:

$$\square(\forall q \in Queue : (P_o(q) \wedge q \neq empty \rightarrow \diamond(\bigvee_{t \in (P_o)^{\bullet}} \bigvee_{p \in t^{\bullet}} (p(q') \wedge first(q) \in q'))))$$

where t^{\bullet} (p^{\bullet}) specifies the set of outgoing places (transitions) of t (p). This property requires that any message in an output event pool is eventually dispatched to input event pool of some instance.

Another way to detect inconsistency is to introduce “invalid” places. A token in invalid places implies the occurrence of inconsistency. This approach is suit to detect contradictions of pre- and post-conditions. In our approach pre- and post-conditions of operations are represented as conditions of Petri net transitions (see Fig. 17). Therefore if there is a operation call without satisfying the precondition, the

operation is not invoked in the model. However, such phenomena – the violation of the precondition, generally indicating the occurrence of inconsistency, is hard to express as properties. But it is easy to be detected by introducing additional places. For example, in Fig. 16, after adding a new item to its content, the post-condition – the total number of items in the cart should be no more than 5 – is tested in the transition *post* and should be satisfied. To detect the violation of the post-condition, a new place *exception* is added to the Petri net derived from activity diagram *addItem*. Then a safety property $\square(\neg exception(\text{"."}))$ is introduced to detect such inconsistencies.

Petri net analysis techniques can also be explored to detect UML inconsistency. This approach is especially suit to check inconsistency between statechart diagrams, activity diagrams and interaction diagrams. The semantics of an interaction is given as a pair of set of traces. The two trace sets represent valid traces T_v and invalid traces T_i . A trace is a sequence of event occurrences. From a system net, we also can extract a set of traces T_s of the system, which is compared with the pair of set of traces T_v and T_i . The following potential results can be obtained from the comparison:

- $T_v \subseteq T_s$, which indicates that the interaction is totally supported by class behaviors.
- $T_s \subset T_v$, which indicates that either behavior of some classes is incomplete or the interaction diagram contains some unnecessary scenarios since there are some traces in T_v not supported by class behaviors.
- $T_v \cap T_s = \emptyset$, which indicates the occurrence of inconsistency. By analyzing each trace in T_s , we can locate the reason of the inconsistency.
- $T_i \cap T_s \neq \emptyset$, which indicates the occurrence of inconsistency. By analyzing each trace in T_s or in the intersection, we can locate the reason of the inconsistency.

In the online shopping example, we first find that $T_v \cap T_s = \emptyset$. The only trace in T_s indicates that the instance *cart* ends in the state *ChangingShippingAddr* while the

instance *customer* ends in the final state. By carefully simulate the trace, we find that the *cart* in the state *AddingItems* receives an unexpected message *setBillingInfo*, which is just ignored since it cannot trigger any transition. Such case happens because the customer first fills in shipping address and then billing information while the cart records these information in the reverse order. This is the inconsistency we are looking for. However, even after correcting such an inconsistency, we still find that $T_s \cap T_i \neq \emptyset$, and the safety property $\Box(\neg exception(\text{"."}))$ is violated in some traces. Some traces in the intersection $T_s \cap T_i$ indicates that a customer can checkout without buying any goods. Other traces in the intersection describe the situation that a customer can buy more than 5 items, which is confirmed by the violation of the property $\Box(\neg exception(\text{"."}))$. The problem is due to the statechart diagrams of class *Customer*, which forces customers to checkout even they buy nothing, and statechart diagrams of class *Cart*, which should ignore or reject additional items.

4.9 Summary

In this chapter, we adopted two-layer AHL-nets as the semantic domain for UML notations. AHL-nets, weaving algebra into Petri nets seamlessly, is good at the description of ADTs and behaviors based on them. Two-layer AHL-nets (component nets and function nets) exploring the idea of “net as token” [152, 153], provides the support for object-oriented concepts, and further separate the model of object behavior from the concern of communication mechanism. The transformation based framework with two-layer AHL-nets as the corner stone, provides an approach to synthesize different UML diagrams into a system net. More specifically, class diagrams are formalized as algebraic class specifications; statechart diagrams are translated into function nets based on associated class specifications. Transitions representing activities in function nets are further refined by AHL-nets translated from associated activity diagrams. Component nets are constructed through the integration of communication mechanisms and functions nets, which are treated as a special kind of

token. Finally, component nets of classes are synthesized into a system net through the application of a set of transformation rules, which are extracted from interaction diagrams. Based on system nets, analysis techniques on Petri nets are explored to detect different kinds of inconsistency. The framework is very flexible. We can construct system net not only for a single scenario, but also for multiple scenarios, which enables us to analyze the relationship between multiple interactions modeled by interaction overview diagrams.

CHAPTER 5

IMPLEMENTATION AND VERIFICATION OF SAM ARCHITECTURE DESIGNS

5.1 Introduction

System nets obtained from the framework illustrated in Chapter 3 are a kind of software architecture models, which can be easily specified by SAM (Software Architecture Model) [160], a architecture description language proposed by Florida International University. SAM is a general formal framework for specifying and analyzing software architectures. The foundation of SAM is a dual formalism combining a Petri net model to define behavioral models and a temporal logic to specify properties.

However, a correct and valid software architecture at design level does not ensure the correctness of its implementation due to the error-prone characteristic of the transformation from a model to its implementation. In order to validate the implementation of a system net, two parts of works have to be done: realizing system models, and verifying or validating the implementation. By constructing the implementation automatically, we can control costs, improve productivity and quality. Although automatic programming from a formal specification is in general impossible [13], generating the implementation from design models automatically is viable since architectural design provides enough details.

In this chapter, we propose a methodology to validate system implementation by combining runtime verification and aspect-oriented programming techniques. The

correctness of SAM contains several concerns [75]. However here we only check if behavioral models satisfy specified properties. To our knowledge, no similar work has been done in other architecture description languages such as MetaH [158], Rapide [100], Unicon [141] and Weaves [61] to verify and validate implementations. Fig. 19 shows the whole picture of SAM Parser – a tool developed to realize and validate SAM designs automatically. The dashed lines indicate the work to be discussed in this chapter. For the implementation of software architecture elements in SAM such as components, connectors and ports, please refer [55].

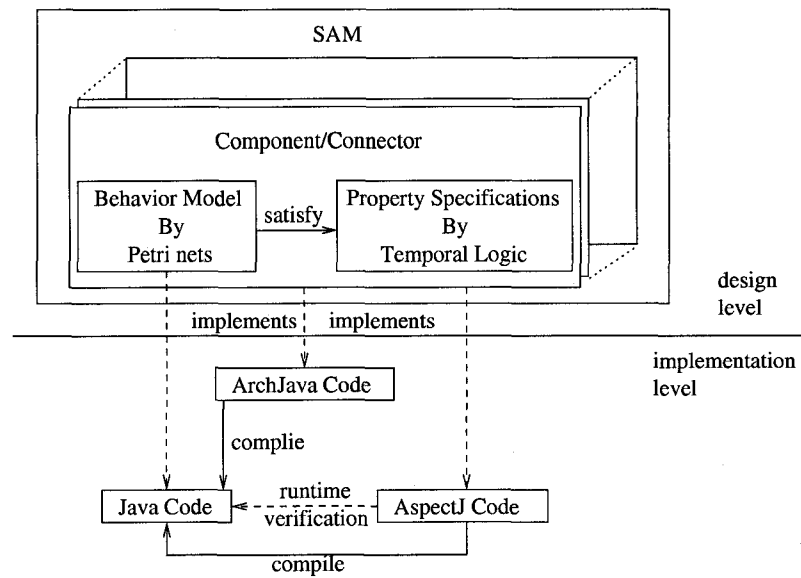


Figure 19: SAM Parser Overview

The rest of this chapter is organized as follows: Section 2 gives a brief introduction of related works. Preliminary knowledge of SAM and the running example are explained in section 3. Section 4 shows the methodology of SAM parser, followed by Petri net implementation and runtime monitor code generation in Sections 5 and 6 respectively. Finally, we show the experiment result and summary.

5.2 Related Works

Currently, some architecture description languages (ADLs) supported the implementation of architectural design in a number of ways [106, 141], but they cannot enforce communication integrity [101, 111] in the implementation that is necessary to enable architectural reasoning about an implementation [2]. By verifying or validating implementations, we can increase our confidence on the correctness and the quality of implementations. This is necessary since “while architectural analysis in existing ADLs may reveal important architectural properties, those properties are not guaranteed to hold in the implementations” [2].

Runtime verification has been proposed as a lightweight formal method applied during the execution of programs. It can be viewed as a complement to traditional methods of proving design model or programs correct before execution. Among the existing works on runtime verification, MaC [97] is the closest to ours. MaC framework needs several inputs from users: a monitoring script in PEDL that provides a mapping between high-level events used in the requirement specification and low-level state information, a requirement specification in MEDL that define properties in a special interval logic, and a system implementation. The monitoring script is used to generate a filter that is a set of program fragments keeping track of monitored objects and sending pertinent state information to the event recognizer, and an event recognizer that detects an event from values of monitored variables received from the filter. Runtime checker, which evaluates requirements over the current event trace received from the event recognizer, is generated from the requirement specification. The MaC framework is proposed to handle any java implementation. However, our work can be viewed as a special case of MaC on software architecture descriptions, more specifically SAM models. Therefore, we can obtain more benefits in terms of automation. In our work, monitoring script and requirement specification is not necessary since they are either implicit or explicit expressed in SAM models. Further more, the system implementation of SAM models is also generated automatically.

Unlike MaC framework, runtime verification systems such as JPAX [69, 70] currently support linear temporal logic, and some analysis algorithms such as Eraser algorithm [138] and deadlock detection algorithm are implemented in the runtime checker too. Further more, JMPaX provides the ability to predict potential safety errors from current successful executions. Currently, our work does not implement such algorithms. However we extend the range of properties to be verified: a subset of first order linear temporal logic formulae. Although this subset looks small, it actually covers most of SAM properties such as response properties involving quantifiers.

Monitoring Oriented Programming (MOP) [33] shows a different way to implement runtime monitoring. MOP is based on the belief that “specification and implementation should together form a system, ... and that they should interact with each other by design rather than grafting monitoring requirements as an add-on to an existing system to increase its safety” [32]. Therefore, requirement logics are inserted into the critical places in the program via annotations by software developers. Actually monitoring code is synthesized automatically from these annotations before compilation and inserted into the appropriate places according to the defined configuration. They support both in-line and out-line, both on-line and off-line monitoring. However, MOP requires that software developers have a deep understanding of the code to catch all “critical” places manually, which is the issue we want to avoid.

Besides runtime verification, there are several other analysis techniques adopted on system implementations to produce a more reliable and error-free software system. Model checking has been applied to check software systems written in Java, C and C++ [12, 30, 68]. Runtime verification focuses on the current program execution, while model checking examines all possible pathes. Unlike testing focusing on the relationship between inputs and outputs, runtime verification underlies the relationship between system implementations and system properties. Therefore, runtime verification is a complement to these techniques, which can also be adopted in our work without difficulty.

5.3 Software Architecture Model

5.3.1 SAM

SAM is an architectural description model based on Petri nets, which are well-suited for modeling distributed systems. SAM has dual formalisms underlying – Petri nets and Temporal logic. Petri nets are used to describe behavioral models of components and connectors while temporal logic is used to specify system properties of components and connectors.

SAM architecture model is hierarchically defined as follows. A set of compositions $C = \{C_1, C_2, \dots, C_k\}$ represents different design levels or subsystems. A set of component C_{m_i} and connectors C_{n_i} are specified within each composition C_i as well as a set of composition constraints C_{s_i} , e.g. $C_i = \{C_{m_i}, C_{n_i}, C_{s_i}\}$. In addition, each component or connector is composed of two elements, a behavioral model and a property specification, e.g. $C_{ij} = (S_{ij}, B_{ij})$. Each behavioral model is described by a PrT net, while a property specification by a temporal logical formula. The atomic proposition used in the first order temporal logic formula is the ports of each component or connector. Thus each behavioral model can be connected with its property specification. A component C_{m_i} or a connector C_{n_i} can be refined to a low level composition C_l by a mapping relation h , e.g. $h(C_{m_i})$ or $h(C_{m_i}) = C_l$. Fig. 20 shows a graphical view of a simple SAM architecture model.

5.3.2 An Example of SAM

Our running example is a coffee machine from [155]. Fig. 21 shows a simplified SAM model of coffee machine. In SAM, there should have a component *CoffeeMachine*, a composition *CoffeeMachine*, and a hierarchical mapping from the component to the composition. However, in order to make the figure more straightforward, we integrate these three parts and still call it composition *CoffeeMachine*. Thus, the composition *CoffeeMachine* has ports that actually belongs to component *CoffeeMachine*. The

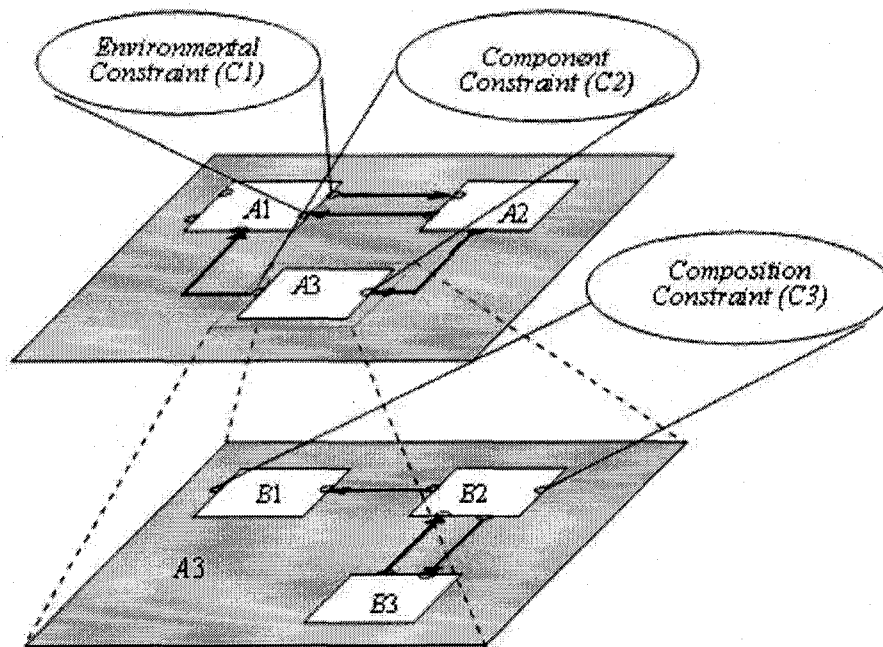


Figure 20: A SAM Architecture Model

connection between a port of the composition and a port of its subcomponent is called glue, which is actually defined in the hierarchical mapping.

From this figure, we can see the coffee machine itself is modeled as a composition *CoffeeMachine*, which has three sub components: *CMInterface*, *CoinHandler*, and *BrewingFacility*, and three connectors: *CH.CMI*, *CH.BF*, and *BF.CMI*. Behavioral models of these components are demonstrated in Fig. 22. The component *CMInterface* acts as the interface of coffee machine to customers. It receives instructions from a customer and transfers them to other parts of the coffee machine. The functionality of the component *CoinHandler* is to make sure that customers have enough money for the specified coffee before the coffee machine serves the customer. The component *BrewFacility* checks the storage of specified coffee and serves the customer if there is enough coffee. The connectors in composition *CoffeeMachine* are very simple: They just transfer messages between a pair of ports in different components.

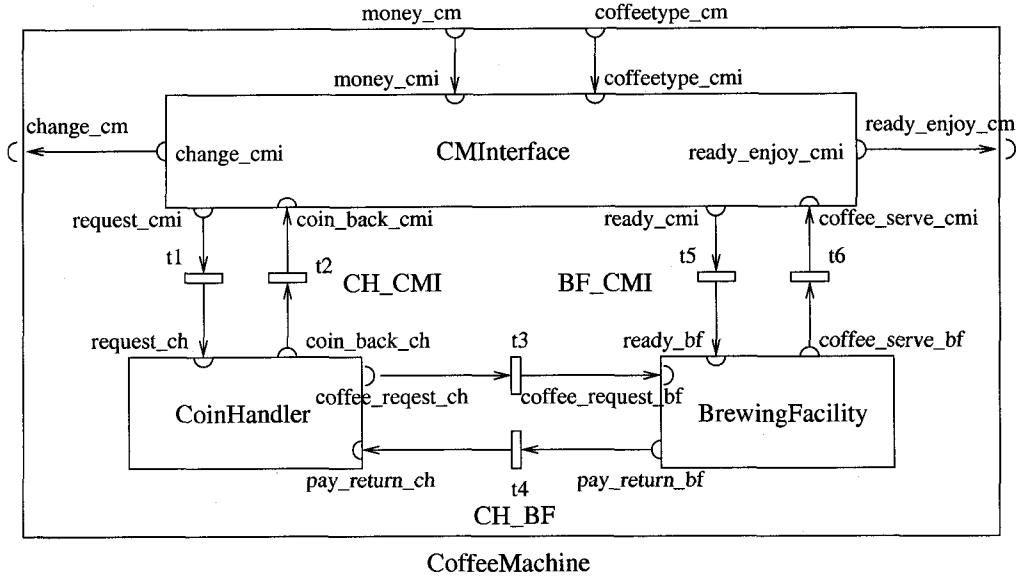
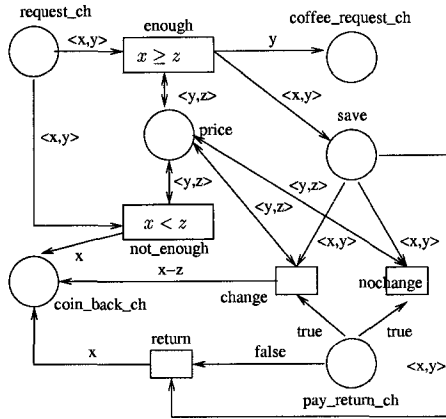


Figure 21: SAM Model of Coffee Machine

Property specifications for each component/connector in SAM are defined by LTL formulae. Some heuristic rules of how to specify temporal properties are given in [75]. The following is a property of component *CoffeeMachine* called *Request*:

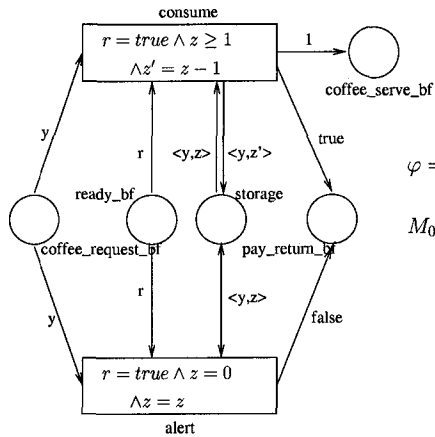
$$\begin{aligned}
 & ((money_cm(85) \wedge coffeetype_cm(2)) \longrightarrow \\
 & \diamond(change_cm(85) \vee (change_cm(10) \wedge ready_enjoy_cm(1)))) \quad (1)
 \end{aligned}$$

In the above formula, atomic predicates are evaluated by checking if a port contains specified messages. For example, atomic predicate $money_cm(85)$ is true if the port $money_cm$ of component *CoffeeMachine* has a message 85. Since in SAM, a port refers to a unique place with the same name in the behavioral model of the component, the atomic predicate also means the place $money_cm$ contains a token 85. Therefore, the above formula specifies the situation that when a user inserts 85 cents to the coffee machine and chooses coffee type 2, the coffee machine either returns 85 cents in case there is not enough coffee, or gives the user 10 cents change and a cup of coffee. Properties can also be expressed as past time linear temporal logic formulae. Formula 2 is the property *RequireMoney* in past time linear temporal logic, where



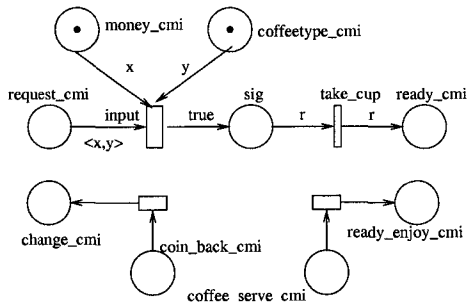
(a) Component CoinHandler

$\varphi = \{request_ch \mapsto \langle int, int \rangle, price \mapsto \langle int, int \rangle, coffee_request_ch \mapsto int, coin_back \mapsto int, save \mapsto \langle int, int \rangle, pay_return_ch \mapsto boolean\}$
 $M_0 = \{request_ch \mapsto \{\}, price \mapsto \{ \langle 1, 50 \rangle, \langle 2, 75 \rangle, \langle 3, 100 \rangle \}, coffee_request_ch \mapsto \{\}, coin_back_ch \mapsto \{\}, save \mapsto \{\}, pay_return_ch \mapsto \{\}$



(b) Component BrewingFacility

$\varphi = \{coffee_request_bf \mapsto int, ready_bf \mapsto boolean, storage \mapsto \langle int, int \rangle, pay_return_bf \mapsto boolean, coffee_serve_bf \mapsto int\}$
 $M_0 = \{coffee_request_bf \mapsto \{\}, ready_bf \mapsto \{\}, storage \mapsto \{ \langle 1, 50 \rangle, \langle 2, 50 \rangle, \langle 3, 50 \rangle \}, pay_return_bf \mapsto \{\}, coffee_serve_bf \mapsto \{\}$



(c) Component CMInterface

$\varphi = \{money_cmi \mapsto int, request_cmi \mapsto \langle int, int \rangle, sig \mapsto boolean, ready_cmi \mapsto boolean, coffeetype_cmi \mapsto int, coffee_serve_cmi \mapsto int, coin_back_cmi \mapsto int, change_cmi \mapsto int, ready_enjoy_cmi \mapsto int\}$
 $M_0 = \{money_cmi \mapsto 75, coffeetype_cmi \mapsto 2, sig \mapsto \{\}, ready_cmi \mapsto \{\}, request_cmi \mapsto \{\}, coffee_serve_cmi \mapsto \{\}, ready_enjoy_cmi \mapsto \{\}, coin_back_cmi \mapsto \{\}, change_cmi \mapsto \{\}$

Figure 22: Behavior of Subcomponents in CoffeeMachine

$\langle * \rangle$ and $[*]$ are the past time operators corresponding to future time operators \diamond and \square in future time linear temporal logic. This formula says there exists an integer m such that whenever a user was served with a cup of coffee by the coffee machine, then the user must have inserted m cents before where $m \geq 50$ if the user chooses coffee type 1, $m \geq 75$ if the user chooses coffee type 2, or $m \geq 100$ if the user chooses coffee type 3.

$$\begin{aligned} & \exists m \in \text{Sort}(\text{money_cmi}), [*](\text{ready_enjoy_cmi}(1) \longrightarrow \\ & \langle * \rangle (\text{money_cmi}(m) \wedge ((\text{coffee_type_cmi}(1) \wedge m \geq 50) \vee \\ & (\text{coffee_type_cmi}(2) \wedge m \geq 75) \vee (\text{coffee_type_cmi}(3) \wedge m \geq 100)))) \end{aligned} \quad (2)$$

5.4 Methodology

Fig. 23 shows the architecture of the methodology. Both SAM models and Petri nets are specified in an XML-based interchange format. For SAM models, a SAM markup language is defined. Petri Net Markup Language (PNML) [21] is used to specify Petri nets. By allowing the definition of Petri net types, PNML supports different versions of Petri nets, such as High Level Petri Nets, Timed Petri Nets, and etc.. Although both SAM and PNML can utilize or specify different versions of Petri nets, here only High Level Petri nets [121] are discussed.

From this architecture, we can see that our work consists of two parts: generating code to execute SAM and Petri nets, and generating monitoring code for run-time verification.

In order to generate code to execute SAM and Petri nets, two sets of classes called templates are predefined to automate the code generation. The template for Petri nets specifies structure and dynamic semantics of high level Petri nets, while the template for SAM describes basic behavior of SAM elements such as components and compositions. It is hard to generate code automatically given a Petri net due to the complexity of sorts, guard conditions of transition and arc labels [98]. Although we cannot achieve this goal for Petri nets in general, we can realize it if the specifications

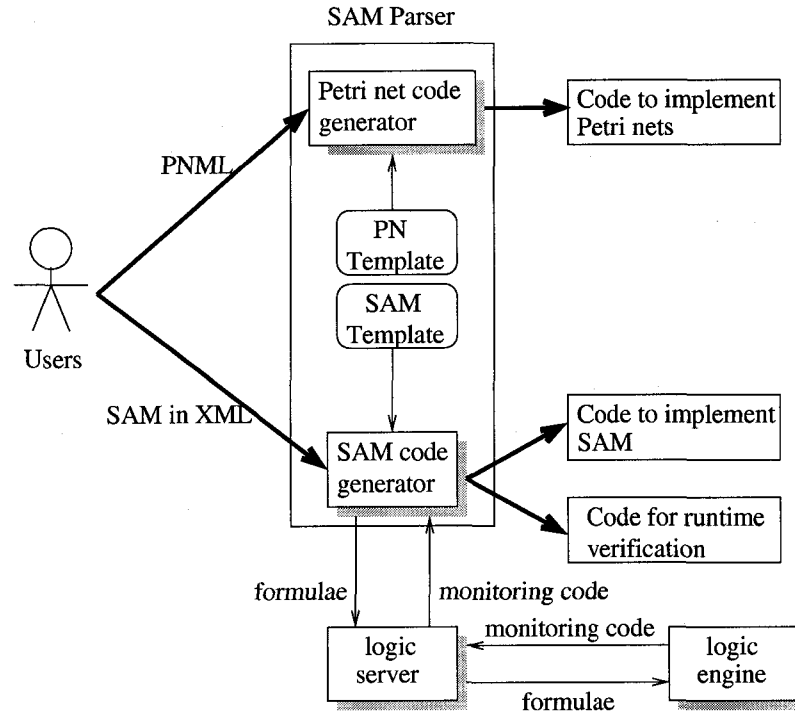


Figure 23: Framework of SAM Parser

of Petri nets satisfies certain restrictions. In our work, we generate Java code to implement Petri nets and ArchJava [2] code to implement SAM since ArchJava is an extension to Java that seamlessly unifies software architecture with implementation and use a type system to ensure that the implementation conforms to architectural constraints.

System requirements are described by temporal logic formulae as a part of SAM components and connectors. For each formula, the monitoring code for runtime verification is generated by logic engine. In order to make the choice of logic independent from SAM parser, a middleware called logic server is inserted between SAM parser and logic engine. In the architecture, a protocol between the SAM Parser and the logic server is defined. Therefore, the choice of logic engine is independent from SAM parser. We use Maude [102] in the current implementation.

The final step is to integrate monitoring code with functionality code. The main concern during the integration is to make sure they can be weaved while they have

clear boundary and do not affect each other's execution. To the best of our knowledge, aspect-oriented programming [63, 113, 122] is the best for our needs since it enables clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols. For each component and connector with a non-empty property specification, an *aspect* [62], defining methods and time to invoke these methods, is generated by integrating monitoring code with time information. In our case, an aspect describes methods to check if properties are satisfied, and defines the appropriate time to invoke these methods— whenever a port sends or receives messages, i.e. a token is added or removed from a place corresponding to a port.

5.5 Implementation of Petri Nets

A behavioral model of a component/connector in SAM is specified by a high level Petri net. Therefore, the implementation of Petri nets is necessary in order to implement SAM automatically. Although lots of works have been done on Petri nets implementation, few of them supports the object-oriented code generation from Petri nets directly.

In order to generate Java code from Petri nets, we predefine a set of Java classes called templates, which specify the structure and dynamic semantics of high level Petri nets. For example, the basic elements of Petri nets such as places, arcs, transitions, guards, inscriptions are defined by individual classes. We also provide dynamic semantics of Petri nets in Java classes *Net* and *Transition*. In other words, we provide a general but maybe not efficient approach to check if a transition is enabled and to be fired.

In our work, we construct a class as a child of templates for each net, place, transition, arc, inscription, initial marking, and guard. The reason for this is to make it easier to understand and maintain. For example, the user can provide a more efficient

way to check the enableness of a transition and the way to fire it by replacing methods of corresponding classes without any side effects on other transitions. The execution of generated code is non-deterministic, i.e. we choose an enabled transition and a valid assignment randomly to fire.

It is hard to generate code automatically given a Petri net due to the complexity of sorts, guard conditions of transition and arc labels [98]. Although we cannot achieve this goal for Petri nets in general, we can achieve it if the specifications of Petri nets satisfy the following restrictions:

- The needed sorts of Petri nets either are Java primitive types such as int, long, and boolean etc., or are defined as a Java classes including its operators, or are a product of already defined sorts.
- The variables occurred in the label of an incoming arc of a transition have the same type as the token sort of the incoming place.
- The variables occurred in the label of an outgoing arc of a transition are defined in the label of an incoming arc of the same transition. In other words, only the label of an incoming arc can define variables.
- If a variable is a product type such as $\text{int} \times \text{int}$ and this product type is generated by Petri net code generator, its field is referred in the form of “.field?”, where ? is the field sequence number starting at 1. For example, x is a variable of type $\text{int} \times \text{int}$, then $x.\text{field1}$ and $x.\text{field2}$ refer to first and second field respectively.

Fig. 24(a) shows a Petri net satisfying the restrictions. The Petri net in Fig. 24(b) violates the restrictions because types of variables $x1$, and $x2$ are not compatible with the type of sort assigned to the corresponding place. The Petri net in Fig. 24(c) violates the restrictions because of variable declaration in an outgoing arc. Therefore, if we choose the specification in Fig. 24(b) and 24(c), manual correction on the generated code is necessary.

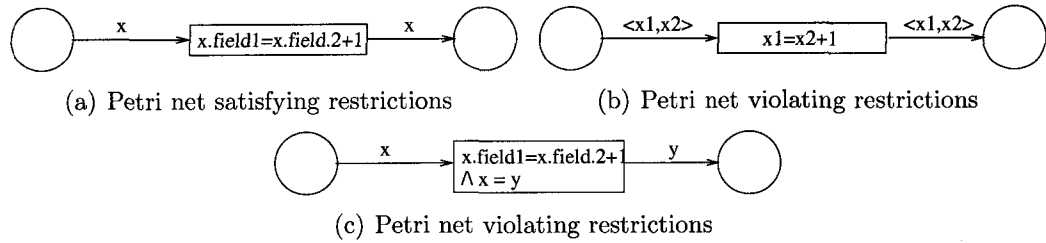


Figure 24: Petri nets satisfying or violating guidance

The SAM Parser can still produce code successfully if the specification of a Petri net does not follow the restrictions. However, the generated code is inexecutable and will produce parse errors before the manual correction.

5.6 Implementation of Run-time Verification

The purpose of runtime verification [132–135] is to monitor, analyze and guide the execution of programs. Traditionally the correctness of a model is verified at design level, runtime verification provides additional correctness assurance at implementation level.

Specific to our case, we need to monitor property specifications for each component/connector during model execution. These property specifications are described as temporal logic formulae. Although SAM can support different temporal logics, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), here we only deal with future time LTL and past time LTL. In order to validate SAM during execution, monitoring code has to be generated for each formula, which is done by the logic server.

By inserting the logic server between the SAM parser and the logic engine, the choice of logic engine is independent from SAM parser. In other words, we can replace one logic with another without any modifications to SAM parser or the code generated by the SAM parser. Currently we choose Maude [102] as our logic engine, and the algorithms to generate code to monitor future time LTL and past time LTL can be found at [131].

There are three different results returned from the execution of the monitoring code of a formula: *true*, *false* and neither true nor false called *unsure*. *True* means the formula is satisfied while *false* means the formula is violated. Generally speaking, the evaluation of a safety property tells us if it is violated; the evaluation of a liveness property tells us if it is satisfied. *Unsure* is an intermediate result, from which we cannot tell if the formula holds or fails. The intermediate result can be returned by the monitoring code of any type of formulae. If the monitoring code of a safety property returns *unsure*, it means the safety property does not fail during the previous execution. For a liveness property, *unsure* generally means it is not true during the previous execution. The following is the monitoring code for Formula 1 in section 5.3.2.

```
private boolean Com_CMInterface$C.F_Request_hasResult = false;
private boolean Com_CMInterface$C.F_Request_result = false;
private int Com_CMInterface$C.F_Request_$state = 1;

public void F_Request(Com_CMInterface$C thisObject) {
    if (thisObject.F_Request_hasResult)    return;

    boolean Pmoney = thisObject.isMessageContained("money_cmi","85");
    boolean Pcoffeetype=thisObject.isMessageContained("coffeetype_cmi","2");
    boolean PgetCoffee=thisObject.isMessageContained("ready_enjoy_cmi","1");
    boolean PmoneyBack = thisObject.isMessageContained("change_cmi","85");
    boolean PgetChange = thisObject.isMessageContained("change_cmi","10");

    switch(thisObject.F_Request_$state) {
        case 1: thisObject.F_Request_$state = PmoneyBack?-1:Pcoffeetype?Pmoney?
                PgetChange ? PgetCoffee ?-1:2: PgetCoffee?3:4:-1:-1 ;
                break ;
        case 2: thisObject.F_Request_$state = PmoneyBack ?-1: PgetCoffee ?-1:2;
                break ;
        case 3: thisObject.F_Request_$state = PgetChange ?-1: PmoneyBack ? -1:3;
                break ;
        case 4: thisObject.F_Request_$state = PmoneyBack ? -1 : PgetChange ?
                PgetCoffee ? -1 : 2 : PgetCoffee ? 3 : 4 ;
                break ;
    }

    if (thisObject.F_Request_$state == -2)
        // The Formula fails : false
    if (thisObject.F_Request_$state == -1)
        // The Formula holds : false
    //Currently, cannot judge the correctness of the formula : unsure
}
```

Although we can generate monitoring code for future and past time LTL, it does not fully satisfy our needs for verifying properties such as first order temporal logic formulae during runtime. First order temporal logic formulae are hard to evaluate against a design model since the domains of quantification variables are generally infinite. However, during the program execution, the number of potential values assigned to a quantified variable is finite, which makes it possible to verify first order temporal logic formulae during runtime by transferring them to temporal logic formulae without quantifications. Due to the complexity of first order temporal logic formulae, we only focus on a subset of them from which monitoring code can be generated automatically by the logic server.

The subset of first order temporal logic formulae we can handle currently has following restrictions:

- Quantification variables are declared before any temporal operators and logic operators. For example, the formula $\forall x \in \text{int}(\Box(p(x)))$ is in the subset, while the formula $\Box(\forall x \in \text{int}(p(x)))$ is not.
- Assignments to all quantification variables occur at the same time slot, and no predicate is evaluated before this time slot. For example, the formula $\forall x \in \text{int}, \exists y \in \text{int}(p(x) \wedge q(y) \longrightarrow \Diamond(r(y)))$ and the formula $\forall x \in \text{int}, \exists y \in \text{int}(p(x) \longrightarrow r(y))$ are in the subset, while the formula $\forall x \in \text{int}, \exists y \in \text{int}(p(x) \longrightarrow \Diamond(r(y)))$ is not since the assignment to y occurs later than the assignment to x .

At first it seems the subset is too small and does not provide enough support for applications. However, due to the characteristics of property specifications on Petri nets and SAM – most of first order temporal logic formulae are response properties [75], the subset can adequately cover the most cases of property specifications in SAM. The following is the core part of code for Formula 2 .

```

final class F_RequireMoney_Helper {
    public int      m;
    public boolean  hasSuccessfulCondition;
    public boolean  isSuccessful = false;
    public boolean  hasFailureCondition;
    public boolean  isFailure = false;
    public boolean[] F_RequireMoney_$pre = new boolean[2];
    public boolean[] F_RequireMoney_$now = new boolean[2];
}

private Vector Com_CMInterface$C.F_RequireMoney_mList = new Vector(5);
private Vector Com_CMInterface$C.F_RequireMoney_classHelperList = new Vector(5);
private boolean Com_CMInterface$C.F_RequireMoney_hasResult = false;
private boolean Com_CMInterface$C.F_RequireMoney_result = false;

public void F_RequireMoney(Com_CMInterface$C thisObject) {
    if (thisObject.F_RequireMoney_hasResult) return;
    Vector mList = thisObject.getMessageFromPort("money_cmi");
    int m;
    for (int i=0; i<mList.size(); i++)
        thisObject.addF_RequireMoney_mList(thisObject,
            ((Integer)mList.elementAt(i)).intValue());
    thisObject.updateF_RequireMoney_classHelperList(thisObject);

    boolean truthValue = true, hasResult = false;
    F_RequireMoney_Helper actElement = null;
    for(int i0=0; i0<thisObject.F_RequireMoney_mList.size(); i0++) {
        m = ((Integer)thisObject.F_RequireMoney_mList.elementAt(i0)).intValue();
        Vector helperClassList = thisObject.F_RequireMoney_classHelperList;
        int j=0;
        for (j=0; j<helperClassList.size(); j++) {
            actElement = (F_RequireMoney_Helper)helperClassList.elementAt(j);
            if( (m == actElement.m) ) break;
        }
        if (j == helperClassList.size()) continue;
        F_RequireMoney$(thisObject, actElement);
        truthValue = true; hasResult = false;
        if ( actElement.hasFailureCondition )
            if ( actElement.isFailure ) {
                truthValue = false; hasResult = true;
            }
        if ( actElement.hasSuccessfulCondition )
            if ( actElement.isSuccessful ) {
                truthValue = true; hasResult = true;
            }
        if (truthValue) break;
    }
    if (hasResult) {
        if(truthValue) The Formula Holds;
        else The Formula Fails;
    } else
        Currently, cannot evaluate the formula;
}

```

As we know, a component/connector in SAM has a property specification, which consists of multiple linear temporal logic formulae (either future time LTL or past time LTL). After gathering monitoring code for each formula from logic server, the SAM parser constructs an aspect for each component/connector. In general, aspects consist of an association of other program entities, ordinary variables and methods, pointcut definitions (interesting points in the execution of a program), inter-type declarations, and advice that declares a time (before, after or around pointcut) to take actions. In the aspect, monitoring code for each formula is invoked to evaluate the formula whenever a message is received from or sent to a port. The following code is an aspect for property specification of component *CMInterface*.

```
public aspect Com_CMInterfaceMonitorAspect {
    pointcut MonitorPoint(): (call(void addMessage(String, Object)) ||
                             call(void removeMessage(String, Object)));
    after(Com_CMInterface$C thisObject) : target(thisObject) &&
                                         MonitorPoint() {
        F_Request(thisObject);
        F_RequireMoney(thisObject);
    }
    ...
    variables and methods generated for each property
    ...
    pointcut ConstructorPoint(): ( !within(SAM_Component) &&
                                   execution(new(...)));
    after(Com_CMInterface$C thisObject) :
        target(thisObject) && ConstructorPoint() {
        //initialize helper variables for each property if necessary
    }
}
```

Although we hope we can check any type of formulae during runtime, there are limitations for runtime verification. In other words, there are some type of formulae that we cannot tell if they hold or fail during runtime. $\Box(p \longrightarrow \Diamond(q))$ is such a formula. We cannot tell if the formula holds since there is an *always* temporal operator, which means the formula should be monitored forever. However, we also cannot tell if the formula fails because of temporal operator *future*, which means you

cannot know the formula fails until the program ends – but at that time there is no monitoring any more.

Similar to model checking, we need a counter example for analysis purpose if a formula fails. To produce a counter example, we record a trace of program execution to a log. The trace from the start to the current spot when the formula fails forms the counter example.

5.7 Experimental Results

We use the coffee machine as the running example. Although it is a little small and simple, all aspects of SAM and Petri nets are covered. For the property specifications, Formulae 1 and 2 are defined in the property specification of component *CMInterface*.

As a result of executing SAM parser, lots of files are generated to implement Petri nets and SAM. Table 4 shows the distribution of generated files, which are the implementations of SAM structure, component/connector behavior, and monitoring codes. From this table, we can see even for this simple example, more than 200 Java classes are generated to implement Petri nets. The reason for this is due to the most important principle for the SAM parser: The generated code is kept simple to understanding and modifying if necessary. In order to implement SAM, one ArchJava file is generated for each component, connector or composition. A component/connector class in ArchJava introduces several Java classes, which are decided by the number of ports contained by the element. In our example, only the property specification of component *CMInterface* is not empty. Therefore, only one aspect is generated, which verifies formulae 1 and 2. One thing we have to point out is that composition *CoffeeMachine* has no behavioral model. Its behavior is decided by its sub-components and sub-connectors.

A log file is used to record the results of execution of generated code including implementation of Petri nets and SAM, and runtime verification. Each step taken by a Petri net is record in the following form:

Table 4: Generated Files for Coffee Machine

<i># of Generated Files</i>	For PN Implementation	For SAM Implementation	For Run-time Verification
CoffeeMachine	0	1	0
CMInterface	11	1	1
CoinHandler	26	1	0
BrewingFacility	14	1	0
CH_CMI	5	1	0
CH_BF	5	1	0
BF_CMI	5	1	0
Templates	14	7	0
Sort	1	0	-
Total	81	14	1

```

<incoming places.{marking}>
    -----component/connector.transition----->
<outgoing places.{marking}>

```

For example, the following step means transition *input* is fired. As a result of the firing, token 85 in place *money_cmi*, and token 2 in place *coffeetype_cmi* are consumed, and token $\langle 85, 2 \rangle$ and 1 are added to place *request_cmi* and place *sig* respectively.

```

<money_cmi={85},coffeetype_cmi={2}>
    -----Transition iutput(input)----->
<request_cmi={⟨85,2⟩}, sig={1}>

```

For the runtime verification, we record the value of each predicate, and the result of the current evaluation. The following is an example of the evaluation of formula 1 named *F_Request*.

Formula CMInterface.F_Request:

```

Pmoney= false
Pcoffeetype= false
PgetCoffee= true

```

PmoneyBack= false

PgetChange= false

Cannot judge Formula F_Request currently!

In the above output, predicates $Pmoney$, $PCoffeetype$, $PgetCoffee$, $PmoneyBack$ and $PgetChange$ refer to $money_cmi(85)$, $coffeetype_cmi(2)$, $ready_enjoy_cmi(1)$, $change_cmi(85)$ and $change_cmi(10)$. From the summary of runtime verification, we can see we cannot judge the correctness of formula 1 and formula 2 .

We know that the original property expressed by the formula 1 is true. After carefully checking the log that records the trace of program, we found that the error was due to the sub-formula $PgetChange \wedge PgetCoffee$ since the program could not guarantee that place $change_cmi$ had token 85 and place $ready_enjoy_cmi$ had token 1 at the same state. Therefore, during runtime verification, we could not assert that the formula was true. On the other hand, due to the \diamond operator, the monitoring code could not return false for the verification of this formula. That is why the verification of the formula 1 returned unsure result. Actually, the formula 1 should be corrected as following:

$$\begin{aligned} & ((money_cmi(85) \wedge coffeetype_cmi(2)) \longrightarrow \\ & (\diamond change_cmi(85) \vee (\diamond change_cmi(10) \wedge \diamond ready_enjoy_cmi(1)))) \quad (3) \end{aligned}$$

For the unsure result of formula 2 , at first it seemed very strange since the purpose of runtime verification was to check if formulae were satisfied or not. However this result is correct because formula 2 is a safety property, which means it has to be true in every state. The unsure result for a safety property tells us that the property did not fail before the current checking point. In this case, it means the formula 2 was true during the lifetime of the coffee machine.

5.8 Summary

Validation and verification of systems at model level is relatively mature, and lots of tools have been developed to support model level verification. However, at implementation level, few work has been done to validate systems. In this chapter, besides generating code automatically to implement SAM and Petri nets, we combine runtime verification and aspect-oriented programming to support the validation of system at implementation level.

Run-time verification has been proposed as a lightweight formal method applied during the execution of programs. It can be viewed as a complement to traditional methods of proving design model or programs correct before execution. Aspect-oriented software engineering [63,113,122] and aspect-oriented programming [49] were proposed to separate concerns during design and implementation. Aspect-Oriented Programming complements OO programming by allowing the developer to dynamically modify the static OO model to create a system that can grow to meet new requirements. In other words, it allows us to dynamically modify models or implementations to include code required for secondary requirements (in our case, it is runtime verification) without modifying the original code. By combining runtime verification and aspect-oriented programming to verify and validate models at implementation level, we can obtain the following benefits:

- The procedure from design models to implementations is generally informal, therefore error-prone. Run-time verification provides a means to validate the procedure indirectly.
- Sometimes, a model cannot be validated or verified at design level. For example, model checking is generally applied to systems with finite states. Unfortunately, the state space of Petri nets can be very huge in many cases. Although different approaches have been proposed to handle state space explosion problem, none

of them has solved the problem. In this case, run-time verification is necessary to increase our confidence on the model.

- Run-time verification can provide a counter example for unexpected exceptions of implementations.
- Run-time verification provides a mechanism to handle exceptions of implementations that are not detected during development.
- By adopting aspect-oriented programming, the code for runtime verification does not affect the functionalities of the code that realizes design models.

CHAPTER 6

CONCLUSION

6.1 Overview

The purpose of this research was to verify and validate UML designs by formalizing and transforming UML diagrams into corresponding parts in the proposed component-based framework, and to develop a tool to realize UML designs automatically for dependability assurance by weaving runtime verification code. This investigation focused on:

1. formal component-based framework in which components are modeled by two-layer algebraic high-level nets and interactions are captured by transformation rules. System models can be constructed by applying transformation rules to component models according to system specifications.
2. UML designs formalization and transformation process that describes how to formalize class diagrams, state machine diagrams, and activity diagrams, and how to extract transformation rules from interaction diagrams.
3. Development of analysis techniques for system models constructed in the framework. In order to explore model checking technique, We developed a set of predicates suitable to describe properties of two-layer algebraic high-level nets.
4. Automated realization of UML designs through system model constructed in the framework, which also weaves into the implementation runtime verification code generated from system properties automatically for dependability assurance.

6.2 Contributions

As stated at the outset, the primary objective of this research was to verify and validate UML designs and provide an approach to realize them for dependability assurance. This section summarizes the contributions and benefits listed in Chapter 1, Table 1.

The primary contribution was the development of a formal component-based framework to model systems, which was described in Chapter 3. The major benefit of the framework is to separate component modeling from interaction modeling, which makes it so flexible that different sub-system, even system models can be constructed by needs according to system specification. Such advantage was accomplished by integrating various theories and techniques, i.e. algebraic specification, algebraic high-level net, category theory, and graph rewriting. More specifically, components and their interactions are modeled by Petri nets and transformation rules, respectively. The (sub)system models can be constructed by applying transformation rules to components according to system specifications.

The next contribution was the approach of transforming UML designs into the framework, which was outlined in Chapter 4. This approach consists of several steps: First, formalizing class diagrams by algebraic specifications. Second, using algebraic specifications to obtain Petri net models from state machine diagrams and activity diagrams. Finally, transformation rules are extracted from interaction diagrams. As a by-product, we provide a precise semantics for class diagrams, state machine diagrams, and activity diagrams.

The following contribution was the development of a process to integrate UML designs into a individual but complete system model by adopting the framework as the semantic domain for UML designs. Various diagrams in UML designs specify different aspects of the system to be built. However, it is hard to obtain a complete overview of the system. The transformation of UML designs into the framework provides one way to solve this problem.

In order to validate and verify UML designs, we adopted model checking and other Petri net analysis techniques to check if a system model satisfies given properties. In order to better utilize model checking tools (Currently Maude), we extended the traditional definition of atomic predicates to fit two-layer algebraic high-level nets better.

The final contribution is the automated implementation of system SAM models described in Chapter 5. Furthermore, in order to provide runtime verification for dependability assurance, this tool can also translate system properties into pieces of monitor code, which are weaved into functionality code through aspect-oriented programming. Therefore, system properties can be checked during program execution.

6.3 Future Work

In the process of this investigation, several areas of research were either identified as natural extensions of this dissertation, or needed further exploration for improvements. These areas are summarized below.

- There are two kinds of properties to be verified in the framework: component behavior property and communication protocol property. Verification of component behavior property generally involves one component, while verification of communication protocol property involves several even the whole system nets, which may be quite large in some situations. To solve this problem, we are investigating several compositional model checking techniques. Among the various proposed automated compositional verification techniques in temporal logic [19, 35, 64] and in Petri nets [85, 154], we found that the interface module technique [19] and the IO graph technique [154] are most relevant to our research. We are currently focusing on how to adapt these compositional verification technique to analyze system nets obtained through our framework. We are also studying compositional temporal logic proving techniques developed in [1].

- In this investigation, inconsistency rules and system properties were given either as plain natural text for general case or as linear temporal logic formula for a specific system. However, the plain natural text is always ambiguous, and the translation from them to formal definition of properties for a specific system model is always a manual process and error-prone. Further investigation is necessary to define inconsistency rules and system properties in UML designs. Object Constraint Language (OCL) coming with UML maybe a good candidate.
- Although an example was shown in each chapter to illustrate the framework, the transformation from UML designs to the framework, and the automated realization of SAM models with runtime verification code, it would be better if this process is applied to a real application covering each steps in the process, i.e. from UML designs to the realization.
- The major motivation of the framework is to verify and validate UML designs. However, I believe this framework can also be explored in other areas for system modeling, such as adaptive software architecture, and agent-oriented software engineering.
- In this research, all analysis work was accomplished based on Petri nets. However, it would be more interesting if we can check the compatibility directly between system models (in the form of Petri nets modeling one or more scenario) and interaction diagrams. Although some work has been done on this area, but they more focused on a simple interaction diagrams without considering complicated artifacts such as iterating, parallel, and non-deterministic execution.

LIST OF REFERENCES

- [1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.
- [2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering, Orlando, FL, USA*, May 2002.
- [3] P. André, A. Romanczuk, et al. An Algebraic View of UML Class Diagrams. In H.Sahraoui and C. Dony, editors, *Acte de la conference LMO'2000*, pages 261–276, 2000.
- [4] M. Andries, G. Engels, et al. Graph Transformation for Specification and Programming. Technical Report 7/96, Universitat Bremen, 1996.
- [5] J. Araujo and A. Moreira. Specifying the Behaviour of UML Collaborations Using Object-Z. In *Americas Conference on Information, Systems (AMCIS)*, 2000.
- [6] F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. *Science of Computer Programming*, 55(1–3):3–52, 2005.
- [7] D. B. Aredo. Formalizing UML Class Diagrams in PVS. In *Proceedings of Workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations at OOPSLA '99, Denver, Colorado, USA*, 1999.
- [8] D. B. Aredo. Semantics of UML statecharts in PVS. In *Proceeding of 12th Nordic Workshop on Programming Theory*, Bergen, Norway, 2000.
- [9] D. B. Aredo. A Framework for Semantics of UML Sequence Diagrams in PVS. *Journal of Universal Computer Science*, pages 674–697, 2002.
- [10] D. B. Aredo, I. Traore, and K. Stolen. Towards a formalization of UML class structure in PVS. Technical Report 272, Department of Informatics, University of Oslo, 1999.
- [11] P. Baldan, A. Corradini, et al. Compositional Semantics for Open Petri Nets based on Deterministic Processes. *Mathematical Structures in Computer Science*, 15(1), 2005.
- [12] T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proceedings of TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 268–283, 2001.

- [13] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, 11(11):1257–1268, 1985.
- [14] R. Balzer. Tolerating Inconsistency. In *Proceedings of 13th International Conference on Software Engineering*, pages 158–165. IEEE Computer Society/ACM Press, 1991.
- [15] L. Baresi and M. Pezzè. On Formalizing UML with High-Level Petri Nets. In *Proceedings of Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 276–304, 2001.
- [16] J. P. Barros and L. Gomes. Actions as Activities and Activities as Petri Nets. In *CSDUML'2003 - Workshop on Critical Systems Development with UML within UML'2003*, 2003.
- [17] J. P. Barros and L. Gomes. A Unidirectional Transition Fusion for Coloured Petri Nets and its Implementation for the CPNTools. In *Proceedings of the 5th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 199–218, 2004.
- [18] T. Basten. *In Terms of Nets : System Design with Petri Nets and Process Algebra*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1998.
- [19] S. Berezin, S. Campos, and E. M. Clarke. Compositional Reasoning in Model Checking. *Lecture Notes in Computer Science*, 1536:81–102, 1998.
- [20] S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 35–45, 2002.
- [21] J. Billington, S. Christensen, et al. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer-Verlag, June 2003.
- [22] C. Bock. Post to the u2p-issues mailing list, 2003.
- [23] E. Boerger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In *Proceeding of 8th International Conference of Algebraic Methodology and Software Technology*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308, 2000.

- [24] E. Boerger, A. Cavarra, and E. Riccobene. Modeling the Dynamic of UML State Machines. In *Proceeding of 8th International Workshop, ASM 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 232–241, 2000.
- [25] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.
- [26] F. Borceux. *Handbook of Categorical Algebra: Basic Category Theory*. Number 50 in *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.
- [27] R. H. Bourdeau and B. H. C. Cheng. A Formal Semantics for Object Model Diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, 1995.
- [28] J. Buck, S. Ha, et al. Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems. *Int. Journal in Computer Simulation*, 4(2), 1994.
- [29] J. Campos and J. Merseguer. On the Integration of UML and Petri Nets in Software Development. *Lecture Notes in Computer Science*, 2006. To appear.
- [30] T. Cattel. Modeling and Verification of sC++ Applications. In *Proceedings of TACAS'98: Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in *Lecture Notes in Computer Science*, 1998.
- [31] M. Chechik, S. Easterbrook, and V. Petrovykh. Model Checking over Multi-Valued Logics. In J. N. Oliveira and P. Zave, editors, *Proceedings of Formal Methods Europe (FME'01)*, 2001.
- [32] F. Chen, M. D'Amorim, and G. Rosu. Monitoring-Oriented Programming: A Tool-Supported Methodology for Higher Quality Object-Oriented Software. Technical Report UIUCDCS-R-2004-2420, University of Illinois at Urbana-Champaign, 2004.
- [33] F. Chen and G. Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
- [34] S. Christensen and L. Petrucci. Towards a Modular Analysis of Coloured Petri Nets. In *Proceedings of 13th International Conference on Application and Theory of Petri Nets*, volume 616 of *Lecture Notes in Computer Science*, pages 113–133, 1992.
- [35] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

- [36] M. Clavel, F. Duran, et al. Maude Manual (Version 2.1). Presented at <http://maude.cs.uiuc.edu/>, April 2004.
- [37] K. Compton, J. Huggins, , and W. Shen. A Semantic Model for the State Machine in the Unified Modeling Language. In *Dynamic Behaviour in UML Models: Semantic Questions, UML 2000 Workshop Proceedings*, pages 25–31, 2000.
- [38] P. P. da Silva. A Proposal for a LOTOS-Based Semantics for UML. Technical Report UMCS-01-06-1, Department of Computer Science, University of Manchester, Manchester, UK, June 2001.
- [39] J. de Lara and H. Vangheluwe. *AToM³* as a Meta-CASE Environment. In *Proceedings of 4th International Conference on Enterprise Information Systems*, 2002.
- [40] Z. Dong, Y. FU, and X. He. Deriving Hierarchical Predicate/Transition Nets from Statechart Diagrams. In *Proceedings of 15th International Conference on Software Engineering and Knowledge Engineering*, 2003.
- [41] Z. Dong and X. He. Integrating UML State-chart and Collaboration Diagrams Using Hierarchical Predicate Transition Nets. In *GI Lecture Notes in Informatics*, 2001.
- [42] S. Easterbrook and M. Chechik. A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints. In *Proceedings of 23rd International Conference on Software Engineering*, pages 411–420, 2001.
- [43] H. Ehrig, G. Engels, et al. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
- [44] H. Ehrig, A. Habel, et al. Parallelism and Concurrency in High Level Replacement Systems. *Mathematical Structures in Computer Science*, 1:361–404, 1991.
- [45] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.
- [46] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer-Verlag, 1990.
- [47] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic High-Level Nets: Petri Nets Revisited. In *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types Joint with the 4th COMPASS Workshop*, volume 785 of *Lecture Notes in Computer Science*, pages 188–206. Springer, 1994.

- [48] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In F. Gaducci and U. Montanari, editors, *Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2002.
- [49] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.
- [50] G. Engels, J. H. Hausmann, et al. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of 3rd International Conference on the Unified Modeling Language*, volume 1939 of *Lecture Notes in Computer Science*, pages 323–337, 2000.
- [51] G. Engels, R. Heckel, and S. Sauer. UML – A Universal Modeling Language? In *ICATPN 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 24–38, 2000.
- [52] R. Eshuis and J. Dehnert. Reactive Petri Nets for Workflow Modeling. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 296–315, 2003.
- [53] L. Favre and S. C. érici. Integrating UML and Algebraic Specification Techniques. In *Proceedings of 32nd International Conference on Technology of Object-Oriented Languages*, pages 151–162, 1999.
- [54] A. C. Finkelstein, D. Gabbay, A. Hunter, et al. Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.
- [55] Y. Fu, Z. Dong, and X. He. A Methodology of Automated Realization of a Software Architecture Design. In *Proceeding of the 17th International Conference on Software Engineering and Knowledge Engineering, Taipei, Taiwan, 2005*.
- [56] M. Gogolla. Graph transformations on the uml metamodel. In *Proceedings of ICALP Workshop Graph Transformations and Visual Modeling Techniques (GVMT'2000)*, pages 359–371, 2000.
- [57] M. Gogolla and F. P. Presicce. State Diagrams in UML: A Formal Semantics using Graph Transformations. In *Proceedings of International Conference of Software Engineering, Workshop on Precise Semantics of Modeling Techniques*, pages 55–72, 1998.

- [58] M. Gogolla and M. Richters. Transformation Rules for UML Class Diagrams. In *Selected papers from the First International Workshop on The Unified Modeling Language; UML'98*, volume 1618 of *Lecture Notes in Computer Science*, pages 92–106. Springer-Verlag, 1999.
- [59] M. Gogolla, P. Ziemann, and S. Kuske. Towards an Integrated Graph Based Semantics for UML. *Electronic Notes in Theoretical Computer Science*, 72(3), 2003.
- [60] J. Gore. *Object Structures: Building Object-Oriented Software Components With Eiffel*. Addison-Wesley Pub, 1996.
- [61] M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering (ICSE13), Austin, TX, USA*, May 1991.
- [62] J. Gradecki and N. Lesiecki. *Mastering AspectJ : Aspect-Oriented Programming in Java*, 2003.
- [63] W. G. Griswold and M. Akit, editors. *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. ACM Press, 2003.
- [64] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
- [65] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4), 293-333 1996.
- [66] J. Hatcliff, X. Deng, et al. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 160–173, 2003.
- [67] J. H. Hausmann, R. Heckel, and S. Sauer. Towards dynamic meta modeling of uml extensions: An extensible semantics for uml sequence diagrams. In *Proceedings of IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01)*, pages 80–87. IEEE Computer Society Press, 2001.
- [68] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.
- [69] K. Havelund and G. Rosu. Java PathExplorer - A Runtime Verification Tool. In *The 6th International Symposium on AI, Robotics and Automation in Space*, 2001.

- [70] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *1st Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, 2001.
- [71] X. He. A Formal Definition of Hierarchical Predicate Transition Nets. In *Proceedings of the 17th International Conference on the Application and Theory of Petri Nets*, volume 1091 of *Lecture Notes in Computer Science*, 1996.
- [72] X. He. Defining UML Class Diagrams using Hierarchical Predicate Transition Nets. In *Proceedings of the Workshop on Defining Precise UML Semantics in ECOOP*, 2000.
- [73] X. He. Formalizing Class Diagrams Using Hierarchical Predicate Transition Nets. In *Proceedings of the 24th International Computer Software and Application Conference*, pages 217–222, 2000.
- [74] X. He. Formalizing Use Case Diagrams in Hierarchical Predicate Transition Nets. In *Proceedings of the IFIP 16th World Computer Congress*, pages 484–491, 2000.
- [75] X. He and Y. Deng. A Framework for Developing and Analyzing Software Architecture Specifications in SAM. *The Computer Journal*, 45(1):111–128, 2002.
- [76] R. Heckel and S. Sauer. Strengthening UML Collaboration Diagrams by State Transformations. In *Proceedings of Fundamental Approaches to Software Engineering (FASE'2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 109–123, 2001.
- [77] B. HNATKOWSKA and Z. HUZAR. Transformation of Dynamic Aspects of UML Models into LOTOS Behaviour Expressions. *International Journal of Applied Mathematics and Computer Science*, 11(2):537–556, 2001.
- [78] C. Hoare. *Communication Sequential Processes*. Prentice Hall, 1995.
- [79] K. Hoffmann, H. Ehrig, and T. Mossakowski. High-Level Nets with Nets and Rules as Tokens. In *Applications and Theory of Petri Nets 2005, 26th International Conference*, volume 3536 of *Lecture Notes in Computer Science*, pages 268 – 288, 2005.
- [80] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

- [81] Z. Hu and S. M. Shatz. Explicit Modeling of Semantics Associated with Composite States in UML Statecharts. *Accepted for publication in the Int'l Journal of Automated Software Engineering*, 2005.
- [82] J. Huggins. Abstract state machine homepage: <http://www.wwcs.umich.edu/gasm>.
- [83] A. Hunter and B. Nuseibeh. Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, October 1998.
- [84] J. B. Jørgensen. Coloured Petri Nets in UML-Based Software Development – Designing Middleware for Pervasive Healthcare. In *Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, 2002.
- [85] E. Y. T. Juan, J. J. P. Tsai, and T. Murata. Compositional Verification of Concurrent Systems Using Petri-net-based Condensation Rules. *ACM Transactions on Programming Languages and Systems*, 20(5):917–979, 1998.
- [86] G. Karsai, S. Neema, et al. A Modeling Language and Its Supporting Tools for Avionics Systems. In *Proceedings of the 21st Digital Avionics Systems Conference*, 2002.
- [87] S.-K. Kim and D. Carrington. Formalizing the UML Class Diagram Using Object-Z. In R. France and B. Rumpe, editors, *Proceedings of UML'99 - The Unified Modeling Language. Beyond the Standard*, volume 1723 of *Lecture Notes in Computer Science*, pages 83–98, 1999.
- [88] S.-K. Kim and D. Carrington. UML Metamodel Formalization with Object-Z: the State Machine Package. Technical Report 00-29, Dept. of Computer Science, the University of Queensland, 1999.
- [89] S.-K. Kim and D. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. In *Proceedings of ZB'2000: Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*, pages 2–21, 2000.
- [90] S.-K. Kim and D. Carrington. Metamodeling Approach to Integrate between the UML State Machine and Object-Z. In *Proceedings of 4th International Conference on Formal Engineering Methods*, volume 2495 of *Lecture Notes in Computer Science*, 2002.

- [91] E. Kindler. A Compositional Partial Order Semantics for Petri Net Components. In *ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets*, volume 1248 of *Lecture Notes in Computer Science*, pages 235–252, 1997.
- [92] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256, 2001.
- [93] S. Kuske, M. Gogolla, et al. An Integrated Semantics for UML Class, Object, and State Diagrams based on Graph Transformation. In M. Butler and K. Sere, editors, *3rd International Conference of Integrated Formal Methods (IFM'02)*. Springer, 2002.
- [94] L. Kuzniarz, G. Reggio, et al., editors. *Workshop on Consistency Problems in UML-based Software Development*. Blekinge Institute of Technology, 2002.
- [95] L. Kuzniarz, G. Reggio, et al., editors. *Workshop on Consistency Problems in UML-based Software Development II*. Blekinge Institute of Technology, 2003.
- [96] J. P. López-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams to Stochastic Petri nets: Application to Software Performance Engineering. *SIGSOFT Software Engineering Notes*, 29(1):25–36, 2004.
- [97] I. Lee, S. Kannan, et al. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
- [98] S. W. Lewandowski and X. He. Generating Code for Hierarchical Predicate Transition Net Based Designs. In *Proceedings of the 12th International Conference on Software Engineering & Knowledge Engineering, Chicago, U.S.A.*, pages 15–22, July 2000.
- [99] J. Lilius. On the Compositionality and Analysis of Algebraic High-Level Nets. *Digital Systems Lab. Series A: Research Report*, (16), 1991.
- [100] D. Luckham, J. Kenney, L. Augustin, et al. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [101] D. C. Luckham and J. Vera. An Event Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9), 1995.

- [102] P. L. M. Clavel, S. Eker and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 1996.
- [103] A. Maggiolo-Schettini and A. Peron. Semantics of Full Statecharts Based on Graph Rewriting. In *Proceedings of Graph Transformation in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 265–279, 1994.
- [104] A. Maggiolo-Schettini and A. Peron. A Graph Rewriting Framework for Statecharts Semantics. In *Proceedings of 5th Int. Workshop on Graph Grammars and their Application to Computer Science*, volume 1073, pages 107–121. Springer-Verlag, 1996.
- [105] K. L. McMillan. Symbolic Model Checking: An Approach to the State Explosion Problem. *Kluwer Academic*, 1993.
- [106] N. Medvidovic, P. Oreizy, et al. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 24–32, 1996.
- [107] J. Merseguer, J. Campos, et al. A Compositional Semantics for UML State Machines Aimed at Performance Evaluation. In *WODES '02: Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES'02)*, pages 295–302, 2002.
- [108] J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, 88(2):105–155, Oct 1990.
- [109] E. Mikk, Y. Lakhnech, et al. Implementing Statecharts in PROMELA/SPIN. In *Proceedings of 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Boca Raton, FL, 1998.
- [110] R. Milner and R. Milner-Gulland. *Communication and Concurrency*. Prentice Hall PTR, 1995.
- [111] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(5), 1995.
- [112] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [113] G. Murphy and K. Lieberherr, editors. *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. ACM Press, 2004.

- [114] K. Narayanaswamy and N. Goldman. “Lazy” Consistency: A Basis for Cooperative Software Development. In *Proceedings of International Conference on Computer-Supported Cooperative Work*, pages 257–264, 1992.
- [115] M. Nielsen, L. Priese, and V. Sassone. Characterizing Behavioural Congruences for Petri Nets. In *Proceedings of 6th International Conference on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 175–189, 1995.
- [116] O. Nierstrasz and F. Achemann. A Calculus for Modeling Software Components. In *Formal Methods for Components and Objects, First International Symposium (FMCO 2002)*, volume 2852 of *Lecture Notes in Computer Science*, 2003.
- [117] B. Nuseibeh, S. Easterbrook, and A. Russo. Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 56(11), November 2001.
- [118] I. Ober. An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions. In *Proceedings of Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop*, volume 2589 of *Lecture Notes in Computer Science*, pages 356–3371, 2003.
- [119] Object Management Group. *Unified Modeling Language*, 1.4 edition, 2001.
- [120] Object Management Group. *Unified Modeling Language*, 2.0 edition, 2004.
- [121] T. C. of ISO/IEC. High-level petri nets - concepts, definitions and graphical notation, iso/iec 15909-1, final committee draft, may 2002.
- [122] H. Ossher and G. Kiczales, editors. *Proceedings of the 1st International Conference on Aspect-oriented Software Development*. ACM Press, 2002.
- [123] S. Owre and N. Shankar. The Formal Semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, aug 1997.
- [124] S. Owre, N. Shankar, et al. *PVS System Guide*. Computer Science Laboratory, SRI International, sep 1999.
- [125] J. Padberg. Place/Transition Net Modules: Transfer from Specification Modules. Technical Report Technical Report 2001-03, TU-Berlin, 2001.
- [126] J. Padberg. Petri Net Modules. *Journal on Integrated Design and Process Technology*, 6(4):121–137, 2002.

- [127] J. Padberg and H. Ehrig. Petri Net Modules in the Transformation-based Component Framework. *Journal of Logic and Algebraic Programming*, accepted, 2005.
- [128] J. Padberg, H. Ehrig, and L. Ribiero. Algebraic High-Level Net Transformation Systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.
- [129] F. Parisi-Presicce and A. Pierantonio. An Algebraic Theory of Class Specification. *ACM Transactions on Software Engineering and Methodology*, 3(2):166–199, 1994.
- [130] L. Priese and H. Wimmel. A Uniform Approach to True-Concurrency and Interleaving Semantics for Petri Nets. *Theoretical Computer Science*, 206(1-2):219–256, 1998.
- [131] G. Rosu and K. Havelund. Rewriting-based Techniques for Runtime Verification. *Journal of Automated Software Engineering*, 2004.
- [132] *First Workshop on Runtime Verification (RV'01), Paris, France. Electronic Notes in Theoretical Computer Science, Volume 55, Issues 2*, 2001.
- [133] *Second Workshop on Runtime Verification (RV'02), Copenhagen, Denmark, Electronic Notes in Theoretical Computer Science, Volume 70, Issues 4*, 2002.
- [134] *Third Workshop on Runtime Verification (RV'03), Boulder, Colorado, USA, Electronic Notes in Theoretical Computer Science, Volume 89, Issues 2*, 2003.
- [135] *Fourth Workshop on Runtime Verification (RV'04), Barcelona, Spain. Electronic Notes in Theoretical Computer Science, Volume 113*, 2004.
- [136] J. Saldhana and S. M. Shatz. UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering*, pages 103–110, 2000.
- [137] J. Saldhana, S. M. Shatz, and Z. Hu. Formalization of Object Behavior and Interactions From UML Models. *International Journal of Software Engineering and Knowledge Engineering*, pages 643–673, 2001.
- [138] S. Savage, M. Burrows, et al. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

- [139] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):1–13, 2001.
- [140] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, sep 1999.
- [141] M. Shaw, R. DeLine, et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), April 1995.
- [142] W. Shen. *The Application of Abstract State Machines in Software Engineering*. PhD thesis, Dept. of EECS, The University of Michigan, 2001.
- [143] C. Sibertin-Blanc. A Client-Server Protocol for the Composition of Petri Nets. In *Proceedings of 14th International Conference on Application and Theory of Petri Nets*, volume 691 of *Lecture Notes in Computer Science*, pages 377–396, 1993.
- [144] C. Sibertin-Blanc. Cooperative Nets. In *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*, volume 815 of *Lecture Notes in Computer Science*, pages 471–490, 1994.
- [145] G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
- [146] G. Spanoudakis and A. Finkelstein. Reconciling Requirements: a Method for Managing Interference, Inconsistency and Conflict. *Annals of Software Engineering*, 3:433–457, 1997.
- [147] G. Spanoudakis and A. Zisman. Inconsistency Management in Software Engineering: Survey and Open Research Issues. *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380, 2001.
- [148] C. Szyperski and R. Vernik. Establishing System-wide Properties of COmponent-based Systems – A Case for Tiered Component Frameworks. In *OMG-DARPA-MCC Workshop on Compositional Software Architectures*, January 1998.
- [149] I. Traoré. An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, 2000.
- [150] K. Turner and M. Sighireanu. *ELOTOS: (Enhanced) Language Of Temporal Ordering Specification*, chapter 10 in *Software Specification Methods*, pages 165–190. Springer-Verlag, 2001.

- [151] R. Valk. Algebraic High-Level Nets.
- [152] R. Valk. Petri Nets as Token Objects - An Introduction to Elementary Object Nets. In J. Desel and M. Silva, editors, *19th International Conference on Application and Theory of Petri nets*, volume 1420 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 1998.
- [153] R. Valk. Concurrency in Communicating Object Petri Nets. volume 2001 of *Lecture Notes in Computer Science*, pages 164–195. Springer-Verlag, 2001.
- [154] A. Valmari. Compositionality in State Space Verification Methods. In *Proceedings of the 17th International Conference on Application and Theory of Petri Nets*, pages 29–56, London, UK, 1996. Springer-Verlag.
- [155] W. M. P. van der Aalst, K. M. van Hee, and R. A. van der Toorn. Component-Based Software Architectures: A Framework Based on Inheritance of Behavior. *Science of Computer Programming*, 42(2-3):129–171, 2002.
- [156] A. van Lamsweerde, R. Darimont, and E. Letier. Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering*, 24(11):908–922, November 1998.
- [157] A. van Lamsweerde and E. Letier. Handling Obstacles in Goal-Oriented Requirements Engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, 2000.
- [158] S. Vestal. Metah user’s manual, 1998.
- [159] E. Y. Wang, H. A. Richter, and B. H. C. Cheng. Formalizing and integrating the dynamic model within OMT. In *Proceedings of the 19th international conference on Software Engineering*, pages 45–55. ACM Press, 1997.
- [160] J. Wang, X. He, and Y. Deng. Introducing Software Architecture Specification and Analysis in SAM through an Example. *Information and Software Technology*, 41(7):451–467, 1999.

APPENDIX A

SIG_{BN}-ALGEBRA

The SIG_{BN}-algebra B for a function net $BN = (N, P_{in}, P_{out}, allocate)$ where $N = (SPEC, X, P, T, type, cond, pre, post, A)$ is given by:

- $B_{Transition} = T$;
- $B_{Place} = P$;
- $B_{InPlace} = P_{in}$;
- $B_{OutPlace} = P_{out}$;
- $B_{Bool} = \{true, false\}$;
- $B_{System} = \{\langle BN, M \rangle \mid M \text{ is a marking of the function net } BN\} \cup \{undef\}$;
- $B_{InEvent} = \cup_{p \in P_{in}} allocate(p)$;
- $B_{OutEvent} = \cup_{p \in P_{out}} A_{type(p)}$;
- $B_{Domain_{x_i}} = A_s$ where $x_i \in X_s$ for $i = 1, \dots, n$;
- $truthValue = true$; $falseValue = false$;
- $enabled_B : B_{System} \times B_{Transition} \times B_{Domain_{x_1}} \times \dots \times B_{Domain_{x_n}} \rightarrow B_{Bool}$ with

$$enabled_B(\langle BN, M \rangle, t, v_{x_1}, \dots, v_{x_n}) = \begin{cases} true : (\bar{\alpha}(pre(t)) \leq M) \\ \quad \quad \quad \text{and } \bar{\alpha}(cond(t)) \\ false : else \end{cases}$$

where α is an assignment for variables in X with $\alpha(x_i) = v_{x_i}$, $i = 1, \dots, n$.

- $enabled'_B : B_{System} \rightarrow B_{Bool}$ with

$$enabled'_B(\langle BN, M \rangle) = \begin{cases} true : \exists t \in B_{Transition}, \\ \quad xi \in Domain_{x_i}, i = 1, \dots, n : \\ \quad enabled_B(\langle BN, M \rangle, t, v_{x_1}, \dots, v_{x_n}) = true \\ false : else \end{cases}$$

- $fire_B : B_{System} \times B_{Transition} \times B_{Domain_{x_1}} \times \dots \times B_{Domain_{x_n}} \rightarrow B_{System}$ with

$$fire_B(\langle BN, M \rangle, t, v_{x_1}, \dots, v_{x_n}) = \begin{cases} \langle BN, M' \rangle & enabled_B(\langle BN, M \rangle, t, \\ & v_{x_1}, \dots, v_{x_n}) = true \\ undef & else \end{cases}$$

where $M' = M \ominus \bar{\alpha}(pre(t)) \oplus \bar{\alpha}(post(t))$ and α is an assignment for variables in X such that $\alpha(x_i) = v_{x_i}$, $i = 1, \dots, n$.

- $hasoutput_B : B_{System} \times B_{OutPlace} \times B_{OutEvent} \rightarrow B_{Bool}$ with

$$hasoutput_B(\langle BN, M \rangle, p, e) = \begin{cases} true : (e, p) \leq M \\ false : else \end{cases}$$

- $hasoutput'_B : B_{System} \rightarrow B_{Bool}$ with

$$hasoutput'_B(\langle BN, M \rangle) = \begin{cases} true : \exists e \in B_{OutEvent}, p \in B_{OutPlace} : \\ \quad hasoutput_B(\langle BN, M \rangle, p, e) = true \\ false : else \end{cases}$$

- $hasinput_B : B_{System} \times B_{InPlace} \rightarrow B_{Bool}$ with

$$hasinput_B(\langle BN, M \rangle, p) = \begin{cases} true : \exists e \in A_{type(p)}(e, p) \leq M \\ false : else \end{cases}$$

- $hasinput'_B : B_{System} \rightarrow B_{Bool}$ with

$$hasinput'_B(\langle BN, M \rangle) = \begin{cases} true : \exists p \in B_{InPlace} : \\ \quad hasinput_B(\langle BN, M \rangle, p) = true \\ false : else \end{cases}$$

- $output_B : B_{System} \times B_{OutPlace} \times B_{OutEvent} \rightarrow B_{System}$ with

$$output_B(\langle BN, M \rangle, p, e) = \begin{cases} \langle BN, M' \rangle : hasoutput_B(\langle BN, M \rangle, p, e) = true \\ undef : else \end{cases}$$

where $M' = M \ominus (e, p)$.

- $input_B : B_{System} \times B_{InEvent} \rightarrow B_{System}$ with

$$input_B(\langle BN, M \rangle, e) = \begin{cases} \langle BN, M \oplus (e, p) \rangle : \exists p \in B_{InPlace} : e \in allocate(p) \\ undef : else \end{cases}$$

Operation *enabled* specifies if a transition is enabled under the current marking and the assignment to variables. Operation *fire* fires a given transition with a given variable assignment. Operation *hasoutput* checks if a given output place contains a given message. Operation *hasinput* checks if a given input place contains a message. Operations *enabled'*, *hasoutput'* and *hasinput'* are the more abstract version of corresponding operations. Operation *output* removes a given message from a given output place, while operation *input* adds a given message to an input place.

APPENDIX B

MAUDE CODE FOR HURRIED PHILOSOPHERS

The following is the functional module for SIG_{BN} -Algebra of Servant in hurried philosophers system mentioned in the Chapter 3:

```
fmod FORK_SYSTEM_NET_SORT is
  including FORKAGENT_PN .
  including SYSTEM_NET_SORT .

  vars    m m1 m2 : Marking .
  var     phil : PhilID .
  var     msg : MsgID .
  var     fork : NzNat .
  var     seat : NzNat .
  var     servant : ServantID .

  op fenabled : System FTrans -> Bool [strat (0)] .
  ceq fenabled([m1 token(AvailFork,seat)
               token(ForkRequest,(phil,servant,MForkRequest,seat))
               token(AvailFork,fork) m2], AssignFork) = true
    if fork = (seat rem ForkSeatNum) + 1 .
  eq fenabled([m],AssignFork) = false [owise] .

  eq fenabled([m1 token(ForkInUse,fork)
               token(ReleasedFork,(phil,servant,MForkRelease,fork))
               m2],RevokeFork) = true .
  eq fenabled([m],RevokeFork) = false [owise] .

  op fenabled : System FTrans PhilID ServantID MsgID NzNat
    -> Bool [strat (6 0)] .
  ceq fenabled([m token(AvailFork,seat) token(AvailFork, fork)
               token(ForkRequest,(phil,servant,MForkRequest,seat))
               ],AssignFork,phil,servant,MForkRequest,seat) = true
    if fork = (seat rem ForkSeatNum) + 1 .
  eq fenabled([m],AssignFork,phil,servant,msg,fork)
    = false [owise] .

  eq fenabled([m token(ForkInUse,fork)
               token(ReleasedFork,(phil,servant,MForkRelease,fork))
               ], RevokeFork,phil,servant,MForkRelease,fork) = true .
```



```

eq fenabled([m],RevokeFork,phil,servant,msg,fork)
    = false [owise] .

op ffire : System FTrans PhilAgentID ForkAgentID MsgID NzNat
    -> System [strat (6 0)] .

ceq ffire([m token(AvailFork,seat) token(AvailFork,fork)
    token(ForkRequest,(phil,servant,MForkRequest,seat))
    ], AssignFork,phil,servant,MForkRequest,seat) =
    [m token(ForkInUse,seat) token(ForkInUse,fork)
    token(ForkAvail,(servant,phil,MForkAvail,seat))
    token(ForkAvail,(servant,phil,MForkAvail,fork))]
    if fork = (seat rem ForkSeatNum) + 1 .

ceq ffire([m token(ForkInUse,seat) token(ForkInUse,fork)
    token(ReleasedFork,(phil,servant,MForkRelease,seat))
    ], RevokeFork,phil,servant,MForkRelease,seat) =
    [m token(AvailFork,fork) token(AvailFork,seat)]
    if fork = (seat rem ForkSeatNum) + 1 .

var p : FPlace .
op favailable : System FPlace -> Bool [strat (1 0)] .

eq favailable([m1 token(ForkAvail,(servant,phil,msg,fork)) m2],
    ForkAvail) = true .
eq favailable([m],p) = false [owise] .

sort    FInEvent FOutEvent .
subsort FInEvent < Message .
subsort FOutEvent < Message .

mb (phil,servant,MForkRequest,fork) : FInEvent .
mb (phil,servant,MForkRelease,fork) : FInEvent .
mb (servant,phil,MForkAvail,fork) : FOutEvent .

op foutput : System FPlace FOutEvent
    -> System [strat (1 3 0)] .
eq foutput([m token(ForkAvail,(servant,phil,msg,fork))],
    ForkAvail,(servant,phil,msg,fork)) = [m] .

op finput : System FInEvent -> System [strat (1 2 0)] .
eq finput([m],(phil,servant,MForkRequest,fork)) =
    [m token(ForkRequest,(phil,agent,MForkRequest,fork))] .
eq finput([m],(phil,servant,MForkRelease,fork)) =
    [m token(ReleasedFork,(phil,servant,MForkRelease,fork))] .
endfm

```

The following is the system module for component architecture Servant in hurried philosophers system mentioned in the chapter 3:

```

mod FORK_SYSTEM_NET_EXEC is
    including QUEUE .

```

```

including FORK_SYSTEM_NET .
including FORK_SYSTEM_NET_SORT .

vars m m1 m2 : Marking .
var msg : MsgID .
vars fork : NzNat .
var phil : PhilID .
var servant : ServantID .
var q : Queue .

crl [Tin_Servant]:
token(FSPin, [(phil,servant,msg,fork)] ; q) token(FSPobject,[m])
=>
token(FSPobject,fininput([m],(phil,servant,msg,fork)))
token(FSPin, q)
if fenabled([m],AssignFork) = false /\
    fenabled([m],RevokeFork) = false /\
    favailable([m],ForkAvail) = false .

crl [TforkAvail]:
token(FSPobject,[m1 token(ForkAvail,(servant,phil,msg,fork)) m2])
token(FSPoutput, q)
=>
token(FSPoutput, q ; [(servant,phil,msg,fork)])
token(FSPobject,
    foutput([m1 token(ForkAvail,(servant,phil,msg,fork)) m2],
            ForkAvail,(servant,phil,msg,fork)))
if favailable([m1 token(ForkAvail,(servant,phil,msg,fork)) m2],
    ForkAvail) = true .

crl [Tresponse_AssignFork]:
token(FSPobject,
    [m1 token(ForkRequest,(phil,servant,MForkRequest,fork)) m2])
=>
token(FSPobject,
    ffire([m1 token(ForkRequest,(phil,servant,MForkRequest,fork)) m2],
          AssignFork,phil,servant,MForkRequest,fork))
if fenabled([m1 token(ForkRequest,(phil,servant,MForkRequest,fork)) m2],
    AssignFork,phil,servant,MForkRequest,fork) = true /\
    favailable([m1 token(ForkRequest,(phil,servant,MForkRequest,fork)) m2],
    AvailFork) = false .

crl [Tresponse_RevokeFork]:
token(FSPobject,
    [m1 token(ReleasedFork,(phil,servant,MForkRelease,fork)) m2])
=>
token(FSPobject,
    ffire([m1 token(ReleasedFork,(phil,servant,MForkRelease,fork)) m2],
          RevokeFork,phil,servant,MForkRelease,fork))
if fenabled([m1 token(ReleasedFork,(phil,servant,MForkRelease,fork)) m2],
    RevokeFork,phil,servant, MForkRelease, fork) = true /\
    favailable([m1 token(ReleasedFork,(phil,servant,MForkRelease,fork)) m2],
    AvailFork) = false .

endm

```

The following is the atomic predicates of component architecture Philosopher in hurried philosophers system mentioned in the chapter 3:

```

mod SIG_BN_PHIL_PREDS is
  protecting PHIL_COMPONENT_ARCH .
  including SATISFACTION .

  subsort    Marking < State .

  var    seat : NzNat .
  var    m : Marking .
  var    phil : PhilAgentID .
  var    table : TableAgentID .

  op  pReading : PhilAgentID -> Prop .
  eq  m token(Reading,phil) |= pReading(phil) = true .

  op  pP3 : PhilAgentID -> Prop .
  eq  m token(P3,phil) |= pP3(phil) = true .

  op  pRequestSeat : Message -> Prop .
  eq  m token(RequestSeat,(phil,table,MSeatRequest,seat))
      |= pRequestSeat((phil,table,MSeatRequest,seat)) = true .

  op  pPhilLeft : Message -> Prop .
  eq  m token(PhilLeft,(phil,table,MPhilLeft,seat)) |=
      pPhilLeft((phil,table,MPhilLeft,seat)) = true .

  op  pThinking : PhilAgentID NzNat -> Prop .
  eq  m token(Thinking,(phil,seat)) |= pThinking(phil,seat) = true .

  op  pP2 : PhilAgentID NzNat -> Prop .
  eq  m token(P2,(phil,seat)) |= pP2(phil,seat) = true .

  op  pEating : PhilAgentID NzNat -> Prop .
  eq  m token(Eating,(phil,seat)) |= pEating(phil,seat) = true .
endm

mod PHIL-PREDS is
  protecting SIG_BN_PHIL_PREDS .
  including SATISFACTION .

  subsort    Marking < State .

  var    seat : NzNat .
  vars   m m1 : Marking .
  var    phil : PhilAgentID .

  op  pPSPBJECT-Reading : PhilAgentID PhilAgentID -> Prop .
  ceq (m1 token(PSPobject, <phil,[m]>))
      |= pPSPBJECT-Reading(phil,phil) = true
  if m |= pReading(phil) = true .

```

```

op pPSPOBJECT-Thinking : PhilAgentID PhilAgentID NzNat -> Prop .
ceq (m1 token(PSPobject, <phil,[m]>))
  |= pPSPOBJECT-Thinking(phil,phil,seat) = true
if m |= pThinking(phil,seat) = true .

op pPSPOBJECT-Eating : PhilAgentID PhilAgentID NzNat -> Prop .
ceq (m1 token(PSPobject, <phil,[m]>))
  |= pPSPOBJECT-Eating(phil,phil,seat) = true
if m |= pEating(phil,seat) = true .

op pPSPOBJECT-P2 : PhilAgentID PhilAgentID NzNat -> Prop .
ceq (m1 token(PSPobject, <phil,[m]>))
  |= pPSPOBJECT-P2(phil,phil,seat) = true
if m |= pP2(phil,seat) = true .

op pPSPOBJECT-P3 : PhilAgentID PhilAgentID -> Prop .
ceq (m1 token(PSPobject, <phil,[m]>))
  |= pPSPOBJECT-P3(phil,phil) = true
if m |= pP3(phil) = true .
endm

```

APPENDIX C

DERIVING HIERARCHICAL PREDICATION TRANSITION NETS FROM UML STATE MACHINES

C.1 HPrTNs

An Hierarchical Predication Transition Net (HPrTN) [71] N consists of (1) a finite hierarchical net structure (P, T, F, ρ) , (2) an algebraic specification $SPEC$, and (3) a net inscription (φ, L, R, M_0) .

(P, T, F) is the essential net structure, where $P \cup T$ is the set of nodes satisfying the condition $P \cap T = \emptyset$. P is called the set of places and T is called the set of transitions. There are two kinds of nodes for both places and transitions - *elementary nodes* (represented by solid circles or boxes) and *super nodes* (represented by dotted circles or boxes). Elementary nodes have the traditional meaning in flat Petri net models. Super nodes are introduced to abstract and refine data and processing in HPrTNs. $\rho : P \cup T \rightarrow \wp(P \cup T)$ is a hierarchical mapping that defines the hierarchical relationships among the nodes in P and T .

The underlying specification $SPEC = (S, OP, Eq)$ consists of a signature $S = (S, OP)$ and a set Eq of S-equations. Signature $S = (S, OP)$ includes a set of sorts S and a family $OP = (Op_{s_1, \dots, s_n, s})$ of sorted operations for $s_1, \dots, s_n, s \in S$. The S-equations in Eq define the meanings and properties of operations in OP . $SPEC$ is a meta-language to define the tokens, labels, and constraints of an HPrTN.

The net inscription (φ, L, R, M_0) associates each graphical symbol of the net structure (P, T, F, ρ) with an entity in the underlying *SPEC*, and thus defines the static semantics of an HPrTN. Thus different HPrTNs have different net inscriptions.

$\varphi : P \cup_{s \in S} (s)$ associate each place p in P with a subset of sorts in S , which defines the valid values for the sort of each place. $L : F \rightarrow Label_S(X)$ is a sort-respecting labeling of N where $Label_S(X)$ (X is the set of sorted variables disjoint with OP) is a set of labels. $R : T \rightarrow Term_{OP, bool}(X)$ is a well-defined constraining mapping of N , which associates each transition t in T with a first order logic formula defined in the underlying algebraic specification. $M_0 : P \rightarrow MCON_S$ is a sort-respecting initial marking of N , which assigns a multi-set of tokens to each place p in P . The tokens of a super place are a sorted union of the tokens of its interface child places since only those tokens are externally accessible.

A marking M of an HPrTN is a mapping $P \rightarrow MCON_S$ from the set of places to multi-sets of tokens. An elementary transition is enabled if its pre-set contains enough tokens and its constraint is satisfied with an occurrence mode. The firing of an enabled elementary transition consumes the tokens in the pre-set and produces tokens in the post-set. A super transition is enabled if at least one of its interface child transitions is enabled and its firing is defined by an execution sequence of its child transitions, and thus its behavior is fully defined by its child transitions. The firing rule of a transition is formally defined in [71]. Two transitions (including the same transition with two different occurrence modes) can fire concurrently if they are not in conflict (the firing of one of them disables the other). Conflicts are resolved non-deterministically. The firing of an elementary transition is atomic, and the firing of a super transition implies the firing of some elementary transition and may not be atomic. We define the behavior of an HPrTN to be the set of all possible maximal execution sequences containing only elementary transitions. Each execution sequence represents consecutively reachable markings from the initial marking, in which a

successor marking is obtained through a step (firing of some enabled transitions) from the predecessor marking.

C.2 Methodology

Before we present the specific rules to formalize different components of state machines, we would like to provide the reader with the methodology that illustrates how to provide a more precise semantics for state machines.

Step 1: Define the way to represent the events and actions associated with transitions. An event is a specification of a type of observable occurrence. Some types of events can have parameters. In HPrTNs, an event instances is realized as a token, which are specified in a uniform format so that they can present any parameters of events.

An action is “a specification of an executable statement that forms an abstraction of a computational procedure resulting in a change in the state of the model” [119]. An action is either synchronous or asynchronous. Since HPrTN can model both of them, only asynchronous actions are considered in my research. Synchronous actions can be transformed into two asynchronous actions. In state machine diagrams, there are several types of actions: *CreateAction*, *CallAction*, *ReturnAction*, *TerminateAction*, *DestroyAction*, *SendAction* and *UninterpretedAction*. The first five actions are modeled as call events sent to the state machine diagram of receiver and *SendActions* are modeled as signal events. For *UninterpretedActions*, we only consider the statements that can be transformed to boolean expressions used in guards of transitions such as assignment statements, if-then-else statements, etc.

Step 2: Formalize all states by individual HPrTNs, called *formal nets*, according to proposed State Rules. In this step, only net structures of *formal nets* are specified. The algebraic specifications and net inscriptions of formal nets are provided during the formalization of transitions, i.e. in Step 3.

- Step 3:* Realize all transitions of state machines by applying different rules related with transitions. The firing of a transition of state machines consists of two actions: leaving all states in *leaving(t)*, and entering all states in *entering(t)*, which are realized by leaving rules and entering rules respectively. As a result, the algebraic specification and net inscriptions of formal nets of states are provided. More over, the individual formal nets are connected due to the realization of transitions.
- Step 4:* Implement the implied mechanisms that are required but not realized in the state machines. Such mechanisms include: 1) event broadcasting: An event instance should be available to the whole system simultaneously; 2) history recording: when a history pseudostate is active, the most recent active substates of the state containing the pseudostate should be active; 3) variable sharing: The actions and guards can share a set of global variables. These mechanisms are critical to understand the dynamic behavior of state machines. One of the main advantages of our methodology is to separate the realizations of state machines and implied mechanisms, since the implied mechanisms maybe different due to the various environments.
- Step 5:* Finally, provide a precise semantics of derived HPrTNs, especially to solve the conflicts introduced by state machines or the procedure of realizations. In addition, we have to establish the relationship between state configurations and markings of HPrTNs to help the understanding of transformation.

Table C.2 illustrates the rules and in which step they will be applied.

C.3 States

In state machines, a state can have five associations: *deferrableEvent*, *entry*, *exit*, *doActivity* and *internalTransition*. *deferrableEvent* specifies a set of event types a

Step	Rules	
	Name	Meaning
2	State Rule	Constructing the formal net of a state
	SynchState Rule	Realizing the synchstates and related transitions
3	Simple Transition: Leaving	Realizing the leaving actions of a simple transition
	Simple Transition: Entering	Realizing the entering actions of a simple transition
	Cross Transition: Leaving	Realizing the leaving actions of a cross transition
	Cross Transition: Entering	Realizing the entering actions of a cross transition
	Group Transition Leaving	Realizing the leaving actions of a group transition
	Group Transition Entering	Realizing the entering actions of a group transition
	Initial State	Realizing the initial pseudostates and related transitions
	History State	Realizing the history pseudostates and related transitions

Table 5: State Machine Diagram Formalization Rules

state machine should retain until an event type is not contained in the *deferrableEvent* of a state configuration ¹ or it triggers a transition. *entry/exit* describes the first/last action whenever the state is entered/exited. *doActivity* lists a sequence of atomic actions that should be executed when the state is active. The activity can be terminated by itself or interrupted when the state is exited. *internalTransition* illustrates a set of transitions that can be fired without exiting or reentering the state.

In the sequel, we assume each state or transition of a state machine has a distinguished name. And each place and transition in an HPrTN, the formal net of the state machine, has a name such as *name1_name2*, where *name1* is the name of the corresponding state or transition in the state machine and *name2* is given by rules during the derivation. *name2* is omitted if the place represents a state vertex. *_name2* specifies a set of places or transitions ending with *_name2* where *name1* can be concluded unambiguous in the context. A token is illustrated in bold font enclosed by double quotation marks.

¹The *deferrableEvent* of a state configuration is the union of the *deferrableEvent* of each state in the state configuration.

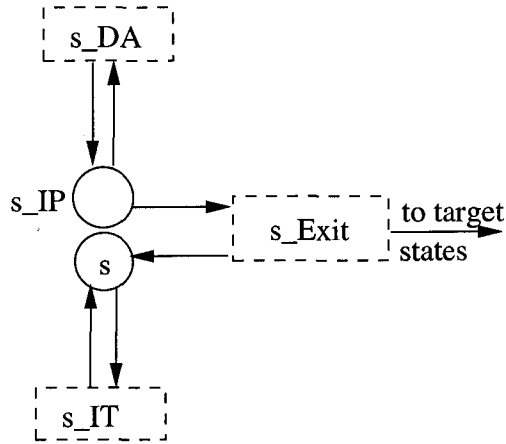


Figure 25: The Formal Net of a Simple State

C.3.1 Simple States

Rule 1 (Simple State) *A Simple state s is realized in Fig. 25:*

In Fig. 25, the super transition s_DA represents the activities described by *doActivity* association and the place s_IP illustrates the interruptible point of the activities. When the activities are finished, the s_IP has a token “FINISHED”; otherwise, it has a token indicating the current step based on interruptible points. The super transition s_IT describes *internalTransition* association of state s . s_DA , s_IP and s_IT , and the associated arcs can be omitted if related associations do not exist. The tokens in elementary place s indicate the status of state s and available event instances that can trigger super transition s_Exit or s_IT . Both s_Exit and s_IT have multiple different enabling conditions. In this paper, we only illustrate these enabling conditions and related firing result and its detail net structure is ignored.

C.3.2 Composite States

Rule 2 (Composite State) *A composite state s is realized by the HPrTN in Fig. 26*

Place s_IP , and transitions s_DA and s_IT are the same as the counterparts in Fig. 25. The tokens in elementary place s indicate the status of state s and available event

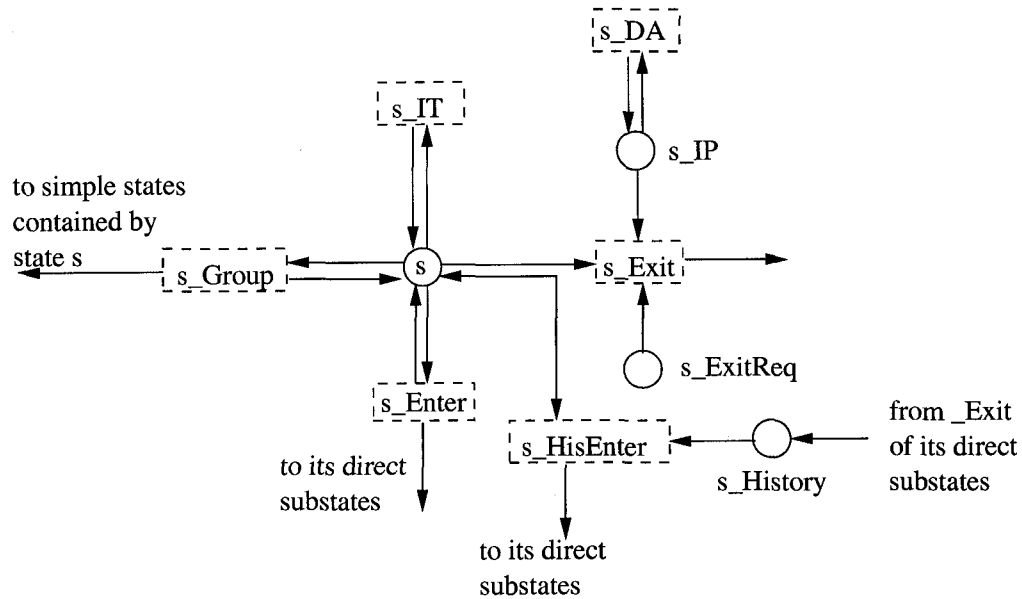


Figure 26: The Formal Net of Composite State

instances that can trigger transition s_Exit or s_Enter . A state can be in one of five statuses, represented by token “dot”, “s_Init”, “s_DHistory”, “s_History”, and “waiting” respectively. Place s can contain at most one of them at any time, and the meanings of them are as follows:

- “dot”: State s is active and idle;
- “s_Init”: State s is active, and it is in the process of entering into its default direct substate;
- “s_DHistory”: State s is active, and it is in the process of entering into its most recent active substates;
- “s_History”: State s is active, and it is in the process of entering into its most recent active direct substates;
- “WAITING”: State s is active, and it is in the process of waiting for the exit of its direct substates of s .

State s is inactive if place s does not contain one of such tokens.

In UML state machines, when a transition t fires, it exits $leaving(t)$ and enters $entering(t)$ automatically. However, such atomic operation cannot be realized by a single transition in HPrTN since the $leaving(t)$ is unknown without giving a state configuration. Therefore, the firing of a transition of state machines is implemented step by step. $s_ExitReq$, containing the exit information of its direct substates, plays an important role in the implementation that is described during the realization of transition of state machines.

$s_History$ and $s_HisEnter$ represents the implementation of the procedure entering into the most recent active direct substates. Since the most recent active direct substates of an and-composite state are its regions, $s_History$ and $s_HisEnter$ can be omitted in an and-composite state. Place $s_History$ contains a token representing the most recent active direct substate or a token “*NULL*” indicating no such information is available and the default state is treated as the most recent active direct substate.

s_Group represents the exit of composite state s and its net structure is explained in the following sections. If there is no transition t such that $source(t) = s$, s_Group can be omitted.

The entry and exit actions of a state are distributed into appropriate $_Enter$ or $_Exit$ transitions during the realization of transitions of state machines.

To focus on the core components of state machines, the $entryAction$, $doActivity$, and $internalTransition$ associations are skipped in the following sections.

C.3.3 Pseudostates

A pseudostate is an abstraction that encompasses different types of transient vertices in a state machine graph. Although pseudostates are transient and have no corresponding status in the object, they enhance the description power of state machines, by making state machines easy to use and understand. UML state machines have seven types of pseudostates: *initial*, *deepHistory*, *shallowHistory*, *join*, *fork*, *junction* and *choice*. In this research, we do not consider history states. The derivation of

initial state is explained in this section, and the derivation of other pseudostates is explained in Section C.4.4.

An *initial* pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state. When the composite state becomes active without specifying which descendant is active, the system enters into the initial state, which then goes into the default state unconditionally.

In state machines, only or-composite states can contain an initial state. When an and-composite state becomes active without specifying the active substates explicitly, the system enters into the initial pseudo states of its regions simultaneously. In order to keep the structure of derivation uniform, we assume each and-composite state contains an initial pseudo state and all regions are its default states.

Rule 3 (Initial State) *Let s be a composite state containing an initial pseudostate. The initial pseudostate is realized by the following enabling condition on transition s_Enter :*

- *If elementary place s contains a token "**s_Init**", s_Enter is enabled. When it fires under this enabling condition, it replaces the token "**s_Init**" by "**dot**" in place s and outputs token "**p_Init**" to the default state p if p is a composite state or "**dot**" if p is a simple state.*

Fig. 27 illustrates an example to realize an initial pseudostate by applying `InitState` rule to a composite state.

C.4 Transitions

A transition is a relationship between two state vertices indicating that the object leaves the source state, enters the target state and performs specific actions when some event instances occur provided that guard condition is satisfied. As a result of firing a transition, some actions will be executed.

Although a transition only has one source state vertex and one target state vertex, when it fires, it may exit from multiple states and enter multiple states, which makes

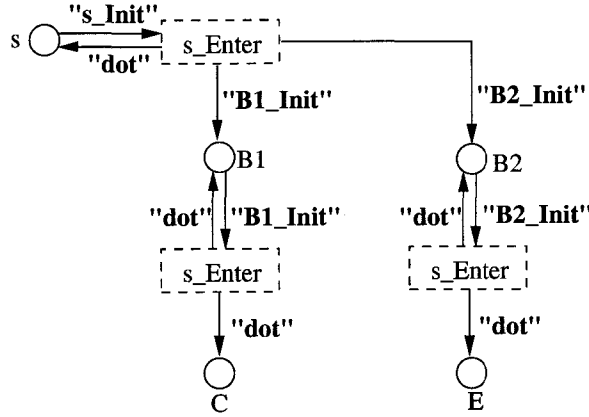


Figure 27: The Formalization of Initial State

the derivation of transitions more difficult. To overcome the difficulty, transitions are classified into four categories: *simple* transitions, *cross* transitions, *group* transitions and *compound* transitions.

Definition 28 (Simple Transition) A transition $t: s_1 \xrightarrow{e[c]/a} s_2$ is a simple transition if and only if $\text{leaving}(t) = \{s_1\}$ and $\text{entering}(t) = \{s_2\}$ ².

Definition 29 (Cross Transition) A transition $t: s_1 \xrightarrow{e[c]/a} s_2$ is a cross transition if and only if there is a composite state s such that $(s_1 \in \text{children}^+(s) \wedge s \in \text{leaving}(t))$ or $(s_2 \in \text{children}^+(s) \wedge s \in \text{entering}(t))$. All such composite states are called *source/target cover states* of t . A *source/target cover state* s is the *outermost source/target cover state* if any other *source/target cover states* are descendants of s .

Definition 30 (Group Transition) A transition $t: s_1 \xrightarrow{e[c]/a} s_2$ is a group transition if and only if s_1 or s_2 is a composite state.

Definition 31 (Compound Transition) A set of transitions is a compound transition if and only if firing all transitions of the set leads the system from a state configuration to another state configuration; and for each transition, either source or target state vertex is a pseudostate.

²In the definitions of simple, cross and group transitions, both s_1 and s_2 must be states, not pseudostates

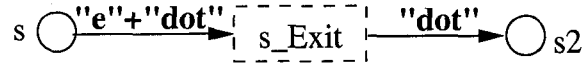


Figure 28: The Formalization of a Simple Transition

However, the categories of the transitions are not disjoint, i.e. a transition can be a cross transition, and a group transition simultaneously. Here, the derivation of each category is explained; then the complex cases are considered.

C.4.1 Simple Transitions

A simple transition leaves one state—source state and enters one state—target state when it fires. The following rule is used to realize a simple transition.

Rule 4 (Simple Transition) *Let transition $t: s_1 \xrightarrow{e[c]/a} s_2$ be a simple transition in a state machine. The transition t is realized by the HPrTN in Fig. 28.*

s_1_Exit is enabled if place s_1 contains a token “dot” and a token “e” representing the occurrence of an event instance e , provided guard c holds. When it fires under such enabling condition, a token “dot” is output to target place s_2 , and the exit action of state s_1 and the entry action of state s_2 are executed sequentially.

However, a transition in state machines may have no trigger event. In such case, it is called a *completion* transition. A *completion* transition is enabled if and only if the activity denoted by *doActivity* association in the source state has been completed provided the guard holds. Thus, in the above rule, if t is a *completion* transition, s_1_Exit may be enabled only if the token contained in s_1_IP is “FINISHED”.

C.4.2 Cross Transitions

When a cross transition fires, it does not only exit (enter) the source (target, resp.) state; but also exits (enters) the source (target, resp.) cover states.

For a cross transition with a source cover state, when it fires, before the source cover state exits, any other active substates of it should abort. Thus the derivation of a cross transition becomes complex since the source cover state has to notify its children

to exit. Also, the *exitActions* of the states should be executed in the order from the innermost substate(s) to the outermost source cover state. In order to modeling such situations, a special event called “**s_Abort**” or “**s_Completion**” for each composite state s is introduced. Such events give the substates an opportunity to abort. The following two rules assume that $source(t)$ and $target(t)$ are simple states.

Rule 5 (Cross Transition Leaving) *Let transition $t: s_1 \xrightarrow{e[c]/a} s_2$ be a cross transition and s be the outermost source cover state of t . Transition t can be realized by adding the following enabling conditions:*

1. *The enabling condition of s_1_Exit is the same as s_1_Exit in Fig. 28; when it fires, a token “**s_Abort**” if t is a completion transition, otherwise a token “**s_Completion**” is output to each simple state that is a descendant of s , and also, a token “ s_2 ” is sent to $s'_ExitReq$ such that $s_2 \in children(s')$.*
2. *For each simple state p such that $p \in children^+(s)$, transition p_Exit is enabled if p contains a token “**dot**”, and either a token “**s_Abort**” or a token “**s_Completion**” (In such case, s_IP should contain a token “**FINISHED**”). When it fires under such enabling condition, a token “**NULL**” is output to $s'_ExitReq$ such that $p \in children(s')$.*
3. *For each composite state p such that $p \in children^+(s)$, following enabling conditions is added to transition p_Exit :*
 - *p_Exit is enabled if $p_ExitReq$ contains a token “**NULL**” (If p is an or-composite state) or n tokens “**NULL**” (If p is an and-composite state). In such case, when p_Exit fires, it output a token “**NULL**” to $s'_ExitReq$ where $p \in children(s')$.*
 - *p_Exit is enabled if $p_ExitReq$ contains a token “**!**” (If p is an or-composite state) or a token “**!**” and $n-1$ tokens “**NULL**” (If p is an and-composite state).*

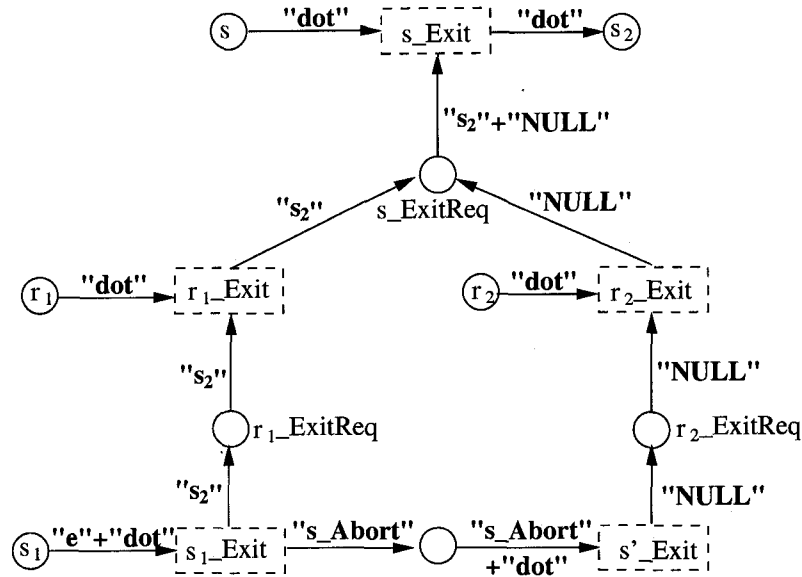


Figure 29: The Formalization of a Cross Transition (Leaving)

state) where l is a place. If p_Exit fires under this enabling condition, it outputs a token " l " to $s'_ExitReq$ where $p \in children(s')$.

4. s_Exit is enabled if $s_ExitReq$ contains a token " l " (If s is an or-composite state) or a token " l " and $n-1$ tokens " $NULL$ " (If s is an and-composite state). When it fires under this enabling condition, it outputs a token " l_Init " (l is a composite state) or a token " dot " (l is a simple state) to place l .

Fig. 29 represents the derivation of a cross transition t with outermost source cover state s .

Actually, enabling conditions 1, 2, and 3 represent the step-by-step exit procedure from innermost substates to the outermost source cover state; and the enabling condition 4 models the procedure of leaving the outermost source cover state and enters into the target state.

In state machines, when a transition fires, ancestor states of the target state and other related states are active at the same time. In our derivation, we realize the procedure step by step.

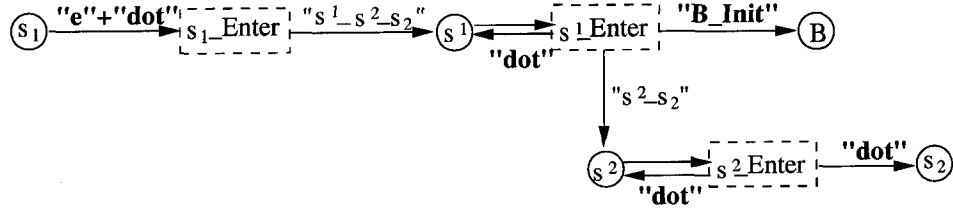


Figure 30: The Formalization of a Cross Transition(Entering)

Rule 6 (Cross Transition Entering) Let transition $t: s_1 \xrightarrow{e[c]/a} s_2$ be a cross transition and s^1, \dots, s^n be the set of target cover states of t such that $s^i \in \text{children}(s^{i-1})$ for any $i \in [2, n]$ and $s_2 \in \text{children}(s^n)$. The enabling condition of s_1_Exit is the same as the enabling condition of s_1_Exit in Fig. 28, and when it fires, a token “ $s^1_s^2 \dots s^n_s_2$ ” is output to place s_1 .

Fig. 30 illustrates a derivation of cross transition t with outermost target cover state s_1 . Generally speaking, for a composite state s , s_Enter is enabled if one of the following conditions holds:

- Place s contains a token “ $s_0_s_1_s_2 \dots s_n$ ” such that $s_0 = s$ and $s_i \in \text{children}(s_{i-1})$ for any $i = 1, 2, \dots, n$. In such case, when it fires, a token “**dot**” is output to place s , and a token “ $s_1_s_2 \dots s_n$ ” is output to place s_1 . If s is an and-composite state, it also sends a token “**dot**” to place s' where $s' \in \text{children}(s)$ and s' is a simple state, or a token “ s'_Init ” to place s' where $s' \in \text{children}(s) \wedge s' \neq s_1$ and s' is a composite state.
- Place s contains a token “**s_p**” such that $p \in \text{children}(s)$. In such case, when it fires, a token “**dot**” is output to place s and place p respectively. If s is an and-composite state, it also sent a token “**dot**” to place s' where $s' \in \text{children}(s)$ and s' is a simple state, or a token “ s'_Init ” to place s' where $s' \in \text{children}(s)$ and s' is a composite state.

C.4.3 Group Transition

Assume transition t is not a cross transition in the following two rules that deal with derivation of group transitions.

Rule 7 (Group Transition Leaving) Let $t: s_1 \xrightarrow{e[c]/a} s_2$ be a group transition, where $children(s_1) \neq \emptyset$ and $children(s_2) = \emptyset$. The transition t is realized by adding the following enabling conditions:

- s_1_Group is enabled if s_1 contains a token “**dot**” and a token “**e**” provided guard c holds. When it fires, it outputs a token “**waiting**” and a token “ s_2 ” to place s_1 , and outputs a token “ s_1_Abort ” if t is not a completion transition, otherwise “ $s_1_Completion$ ” to all simple states that are descendants of state s_1 .
- For each simple state p such that $p \in children^+(s_1)$, enabling condition of p_Exit triggered by token “**_Abort**” or “**_Completion**” is the same as the enabling condition 2 in Cross Transition Leaving Rule;
- For each composite state p such that $p \in children^+(s_1)$, the enabling condition 3 in Cross Transition Leaving Rule is also added to transition p_Exit ;
- s_1_Exit is enabled if s_1 contains a token “**waiting**” and a token “ s_2 ”, and $s_1_ExitReq$ contains enough “**NULL**” tokens (If s_1 is an and-composite state, enough means each direct substate contributed a “**NULL**” token. If s_1 is an or-composite state, enough means one). When it fires in such case, it outputs a token “**dot**” to place s_2 .

Fig. 31 describes the derivation of group transition $t: s_1 \xrightarrow{e[c]/a} s_2$.

Rule 8 (Group Transition Entering) Let $t: s_1 \xrightarrow{e[c]/a} s_2$ be a group transition, where $children(s_2) \neq \emptyset$ and $children(s_1) = \emptyset$. Such a transition is realized by adding the following enabling condition:

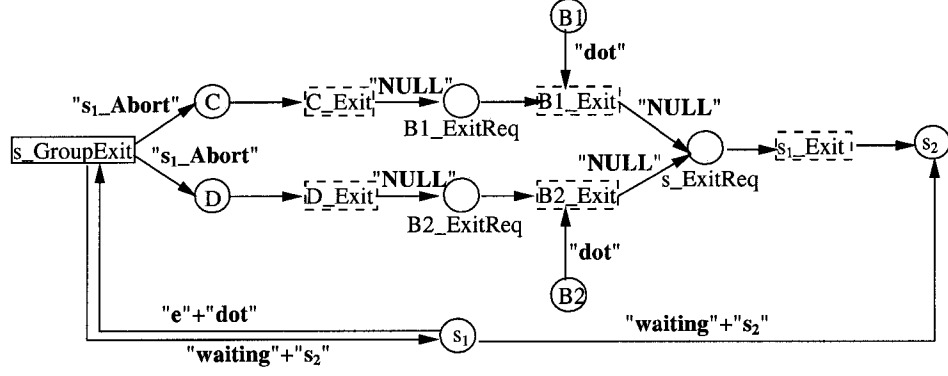


Figure 31: The Formalization of a Group Transition(leaving)

- s_1_Exit is enabled if place s_1 contains a token “dot” and a token “e” representing an event instance e . When it fires, a token “ $s^1_s^2_ \dots_s^n$ ” is output to place s^1 where for any $i = 2, \dots, n$, $s^i \in children(s^{i-1}) \wedge s^n = s_2 \wedge (\exists states' : s_1 \in children(s') \wedge s^1 \in children(s'))$.

As we said before, a transition $t: s_1 \xrightarrow{e[c]/a} s_2$ can be a cross transition and a group transition simultaneously. Thus t can be one of the following cases:

- $children(s_1) \neq \emptyset$ and s is the outermost target cover state of transition t : Such case can be solved by applying Group Transition Leaving Rule, then Cross Transition Entering Rule;
- $children(s_1) \neq \emptyset$ and s is the outermost source cover state of transition t : This case can be solved by applying Group Transition Leaving Rule first. Then treating s_1 as a simple state and apply Cross Transition Leaving Rule;
- $children(s_2) \neq \emptyset$ and s is the outermost source cover state of transition t : Such case can be solved by applying Cross Transition Leaving Rule, then Group Transition Entering Rule;
- $children(s_2) \neq \emptyset$ and s is the outermost target cover state of transition t . Such case can be solved by applying Cross Transition Entering Rule with a

minor modification: when s_1_Exit is fired, a token “ $s^0_s^1_s^2_ \dots s^n$ ” such that $s^0 = s \wedge \forall i = 1, \dots, n : s^i \in children(s^{i-1}) \wedge s^n = s_2$, is output to s .

C.4.4 Compound Transition

A transition connects two state vertices, but maybe one or both state vertices are transient pseudostates. These transient state vertexes include *fork*, *join*, *junction*, and *choice*, and *history* formalized in Sec. C.3.3.

A compound transition consists of multiple sets of transitions that should be fired sequentially. If a set of transitions of a compound transition is fired, it is guaranteed that the next set of transitions should be fired since a state machine cannot be ”stuck” at some transient state vertices. Such situations are hard to formalize since HPrTNs cannot predicate if a transition is enabled. However, some simple compound transitions are easy to handle.

Unlike the classic *Statechart*, some constraints are imposed on the compound transitions in state machines for practical reasons. Some of them affecting our derivations are listed below:

- If the source state vertex of a transition is a *fork* pseudostate, then the target state vertex must be a state and the transition cannot have guards and triggers;
- If the target state vertex of a transition is a *join* pseudostate, the source state vertex must be a state and the transition cannot have guards and triggers.

The transition cannot have triggers if the source state vertices are pseudostates.

Rule 9 (Compound Transition:Join) *Let a set of states s_1, \dots, s_m be concurrent*³. *For each state s_i , there is a transition $t_i: s_i \xrightarrow{/a_i} s$ from s_i to the same join pseudostate s . And there is a transition $t: s \xrightarrow{[c]/a} s'$. Assume transitions t_1, \dots, t_m*

³A set of states is concurrent if and only if any two of them are not ancestrally related and they can appear in a state configuration.

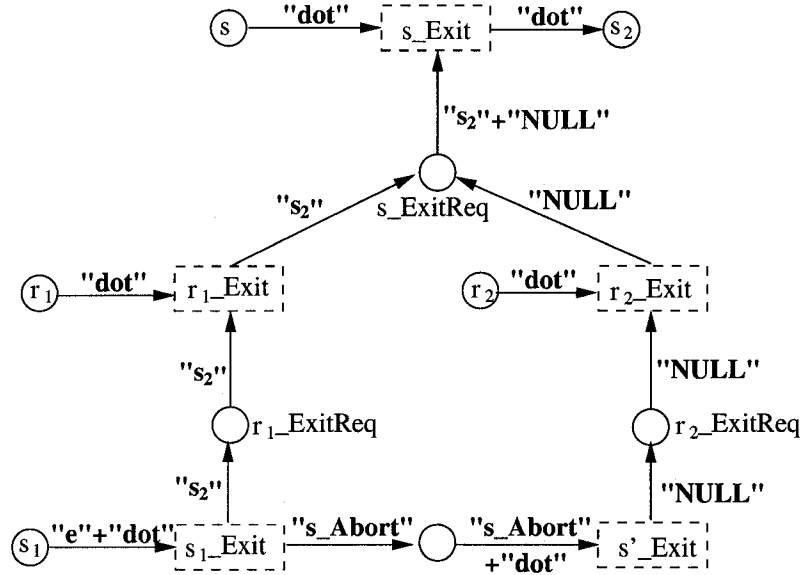


Figure 32: The Formalization of a Compound Transition(Join)

share an outermost source cover state s_{osc} ⁴. Thus, this compound transition can be handled in the following way:

- Transitions t_1, \dots, t_m are represented by a group transition: $t': s_{osc} \xrightarrow{[e]/a_1 \wedge \dots \wedge a_n}$
 s Thus, we apply the Group Transition Rules on t' .
- Pseudostate s is treated as a simple state. Thus Simple State Rule is applied;
- Applying appropriate rules to the transition t .

Fig. 32 delineates the statechart diagram and corresponding HPPrTN by applying Compound Transition Join Rule.

However, we have to guarantee that when s_{osc_Group} is enabled, state s_1, \dots, s_n is active. This can be done by obtaining current state configuration from place Current, which is explained in next section.

A compound transition can be enabled only if a state machine can leave a state configuration and enter another state configuration. In other words, a state machine

⁴If they do not share an outermost source cover state, there is a transition among them such that its source and target states are concurrent. Such a transition is not structured, and not encouraged in UML.

cannot be stuck in a pseudostate. To justify whether a compound transition is enabled or not, all the guard conditions along a compound transition should be evaluated at the beginning. Thus the guard c is evaluated in transition $s_{osc-Group}$. However, when a state machine enters into a pseudostate, it may execute some actions (in this case, it is a_1, \dots, a_m) that change the truth value of guard c . In order to find such exception, the guard c is reevaluated in s_Exit .

Rule 10 (Compound Transition:Fork) *Let a set of states s_1, \dots, s_m be concurrent. For each state s_i , there is a transition $t_i: s \xrightarrow{a_i} s_i$ where s is a fork pseudostate. And there is a transition $t: s' \xrightarrow{e[c]/a} s$, where s' is a state vertex. Such case can be handled in the following ways:*

- *Fork pseudostate s is treated as a simple state and Simple State Rule is applied;*
- *Applying appropriate rules to transition t ;*
- *Transitions t_1, \dots, t_m are represented by a enabling condition of transition s_Exit . s_Exit is enabled if place s contains a token. When it fires, it outputs appropriate tokens to each place s_i according to corresponding entering rules.*

Rule 11 (Compound Transition:Junction) *Let s_1, \dots, s_m be a set of states; s a junction pseudostate and s'_1, \dots, s'_k a set of states. There are transitions such that $t_i: s_i \xrightarrow{e_i[c_i]/a_i} s$, $i = 1, \dots, m$; and transitions such that $t'_j: s \xrightarrow{[c'_j]/a'_j} s'_j$ $i = 1, \dots, k$. Such case can be formalized in the following ways:*

- *Junction pseudostate s is treated as a simple state and Simple State Rule is applied;*
- *Applying appropriate rules to each transition t_i ($i = 1, \dots, m$) and t'_j ($j = 1, \dots, k$).*

In order to avoid that a state machine is stuck in a junction pseudostate, the guard of s_i_Exit is changed to $c_i \wedge (c'_i \vee \dots \vee c'_k)$ to delineate the enabling condition of the compound transition. However, a state machine can also be stuck in pseudostate s if action a_i affects the evaluation of the following conditions. In such case, place s contains a token that cannot be consumed.

Rule 12 (Compound Transition:Choice) *Let s_1, \dots, s_m be a set of states; s a state and s' a choice pseudostate. There are transitions such that $t_i: s' \xrightarrow{[c_i]/a_i} s_i$, $i = 1, \dots, m$; and a transition $t: s \xrightarrow{e[c]/a} s'$. Such case can be formalized in the following ways:*

- *Choice pseudostate s is treated as a simple state and Simple State Rule is applied;*
- *Applying appropriate rules to each transition $t_i (i = 1, \dots, m)$ and t .*

As the same reason explained in join and junction rules, the guard of s_i_Exit is changed to $c \wedge (c_1 \vee \dots \vee c_m)$.

A simple compound transition can be formalized using the above rules. For the complicated compound transitions that contain two or more pseudostates, we have to calculate the enabling condition at the beginning. Such calculation can be found in [65].

VITA
ZHIJIANG DONG

April 20, 1973 Born, P.R.China

1990–1994 B.S., Applied Mathematics
Huazhong University of Science and Technology
Wuhan, P.R.China

1994–1997 M.S., Computer Science
Huazhong University of Science and Technology
Wuhan, P.R.China

1998–1999 Software Engineer
Everbright Securities, Ltd., Co.
Shenzhen, P.R.China

2000–2006 Doctoral Candidate in Computer Science
Florida International University
Miami, Florida

Teaching Assistant
School of Computing and Information Sciences
Florida International University
Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Z. Dong, Y. Fu, and X. He. *A Framework for Component-based System Modeling* in the 18th International Conference on Software Engineering and Knowledge Engineering. San Francisco Bay, USA, July 5 – 7, 2006.

Z. Dong, Y. Fu, and X. He. *UML Consistency Checking Through Transformation based Petri Net Framework* in the workshop of Consistency in Model Driven Engineering (CoMoDE), in conjunction with European Conference on Model Driven Architecture, Nuremberg, Germany. November 7 – 10th, 2005.

Z. Dong, Y. Fu, and X. He. *An Integrated Runtime Monitoring Framework for Software Architecture Model Verification* in the 9th IASTED International Conference on Software Engineering and Applications. Phoenix AZ, USA. November 14 – 16, 2005.

Z. Dong, Y. Fu, and X. He. *Automated Runtime Validation of Software Architecture Design*. Lecture Notes in Computer Science, vol. 3816, pages 446 – 457. 2nd International Conference on Distributed Computing & Internet Technology. Bhubaneswar, India. December 22 – 24, 2005.

- Z. Dong, Y. Fu, and X. He. *Deriving Hierarchical Predicate/Transition Nets from Statechart Diagrams*. In proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering, pages 150 – 157. San Francisco CA, U.S.A. July 1 – 3, 2003.
- Z. Dong and X. He. *Integrating UML State-chart and Collaboration Diagrams Using Hierarchical Predicate Transition Nets*. Lecture Notes in Informatics, vol. P-7, pages 99 – 112. 2001.
- Y. Fu, Z. Dong, X. He. *Modeling, Validating and Automating Composition of Web Services* in the 2006 IEEE International Conference on Web Engineering (ICWE 2006), California, 2006.
- Y. Fu, Z. Dong, X. He. *Formalizing and Validating UML Architecture Descriptions of Web Systems* in the Workshop on Model-Driven Web Engineering, in conjunction with ICWE 2006, California, July 11, 2006.
- Y. Fu, Z. Dong, X. He. *An Approach to Web Services Oriented Modeling and Validation* in the 2006 International Workshop on Service Oriented Software Engineering, in conjunction with International Conference on Software Engineering (ICSE'06). Shanghai, China, May 27–28, 2006.
- Y. Fu, Z. Dong, and X. He. *An Approach to Validation of Software Architecture Model* in the 12th Asia-Pacific Software Engineering Conference, Taipei, Taiwan. December 15 – 17, 2005.
- Y. Fu, Z. Dong, and X. He. *A Methodology of Automated Realization of a Software Architecture Design*. In proceedings of 17th International Conference on Software Engineering and Knowledge Engineering, pages 412 – 417. Taipei, Taiwan. July 14 – 16, 2005.