Diagnosis of Large Software Systems Based on Colored Petri Nets

Yingmin Li

To cite this version:

Yingmin Li. Diagnosis of Large Software Systems Based on Colored Petri Nets. Software Engineering [cs.SE]. Université Paris Sud - Paris XI, 2010. English. NNT : . tel-00551301

HAL Id: tel-00551301
https://theses.hal.science/tel-00551301

Submitted on 3 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

UNIVERSITÉ PARIS SUD 11

présentée pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PARIS-SUD 11

Spécialité : INFORMATIQUE

Par

YINGMIN LI

Diagnosis of Large Software Systems Based on Colored Petri Nets

Thèse soutenue le 9 Décembre 2010 devant le jury composé de:

M.	Albert Benveniste, Directeur de recherche, INRIA/INSIA	Rapporteur
M.	Luca Concole, Professeur, Université Turin, Italie	Rapporteur
Mlle.	Fatiha Zaidi, Maître de conférence, Université Paris sud	Examinatrice
M.	Serge Haddad, Professeur, ENS Cachan	Examinateur
M.	Phillipe Dague, Professeur, Université Paris sud	Directeur de thèse
M.	Tarek Melliti, Maître de conférence, Université Evry	Encadrant de thèse

Laboratoire de Recherche en Informatique, INRIA Saclay-Île-de-France, U.M.R. CNRS 8623,

Université Paris-Sud, 91405 Orsay Cedex, France

Preface

This thesis is the final work of my Ph.D. study at the Laboratory of the Informatics Research (LRI), University of Paris South, XI, France. It serves as documentation of my work during the study, which has been made from autumn 2006 until spring 2010. The study has been funded by the EU through the FP6 IST project 516933 WSDIAMOND (Web Services DIAgnosability, MONitoring and Diagnosis) [138], national ANR (Agency of National Research) project WEBMOV (Web services Modeling and Verification), national ANR project DIAFORE (DIAgnostic of the FOnctions REparation), and national ANR project Docflow (analysis, monitoring, and optimization of Web documents and services).

The thesis consists of seven chapters. Four chapters (three to six) are the integration of the papers that are accepted by or intended for an international journal or proceedings. These chapters illustrate a research of the CPN model-based diagnosis. The experiments and conclusion are given in the later chapters. The first and second chapters give a general introduction to model-based diagnosis and a survey of the new scientific results related to this thesis.

Acknowledgments

I wish to take this opportunity to express my gratitude to my supervisor professor Philippe Dague for introducing me to the area of Discrete Event System (DES) diagnosis, patiently instructing for the meaningful direction, and formulating the main working hypothesis for this thesis, although he allowed me the freedom to wander off. Thanks, Philippe for good support, advice, and encouragement through ups and downs all those years.

I also want to express my appreciation to the co-supervisor Dr. Tarek Melliti. He spent a lot of time to work with me, inspired me for the new research direction and approach, and always gave me a hand when I was stacked.

I¡¯m grateful for the financial support from the research project WSDIAMOND, WebMov, Diafore, and Docflow.

A number of colleagues and fellow students, although most of them have left the laboratory, have greatly contributed to good inspiring working and social environment; especially Mr. Omar Aaouatif, Winwise company, for developing the monitoring platform - thanks. Special thanks for always helping out goes to our laboratory assistant Marie Dominique, celine Halter, Geneviève Sabater, and Martine lelievre.

I also wish to thank Dr. Yuhong Yan for the in-deep and fruitful discussion on the automata diagnosis and diagnosability properties.

My sincere thanks go to my mother Yuehua Hao and brother Yingqing Li for their deep love and sincere support for me. My love goes on with you.

Contents

Preface 2

Acknowledgments 3

List of Tables 9

List of Figures 11

List of Definitions 1

1 Introduction 1

1.1 Abnormal behavior of software system 3
1.1.1 Communication between the components 3
1.1.2 Data and activities flow 4
Data transforming activities 4
Dependency of Data 4
1.1.3 Faulty state 5
1.2 Problematic of MBD for software systems 6
1.2.1 The choice of abstract model 6
1.2.2 The observation for diagnosis 6
Imperfectness of observations 7
Exception assertion 7
1.2.3 Minimal diagnosis 8
1.3 Our contribution 8
1.3.1 Model construction 9
1.3.2 Diagnosis approach 9

1.3.3 Decentralized topology 10
1.3.4 Application: WSDIAMOND project [138] 10

2 Model based diagnosis of discrete event systems **12**
2.1 Introduction 12
2.1.1 Model based diagnosis 12
2.1.2 Discrete event system 13
2.1.3 Dining philosophers example 14
2.2 DES models 14
2.2.1 Labeled transition system and automata 15
2.2.2 Petri nets 17
2.3 Modeling diagnosis with DES 19
2.3.1 Fault representation 19
2.3.2 Observation 24
Observation absence 24
Partially ordered observation 25
2.3.3 Diagnosis of DES 25
2.4 Diagnosis methods 26
2.4.1 Diagnoser 26
2.4.2 PN unfolding 28
2.4.3 PN backward reachability analysis 30
2.5 Architecture of DES diagnosis 31
2.5.1 Centralized diagnosis 32
2.5.2 Decentralized diagnosis 32
2.5.3 Distributed diagnosis 35
2.6 Conclusion 37

I Theory **40**

3 Colored Petri net model for MBD **42**
3.1 Introduction 42
3.2 Structure and dynamic 43
3.2.1 Structure of CPN 43

3.2.2 Dynamic of CPN 48
3.3 CPN as a fault model for software systems 50
3.3.1 The CPN fault model structure 51
Places types: data status 51
Arcs expressions: abstract data dependency 51
Transition modes: faults 52
CPN fault model definition 53
3.3.2 The CPN fault model dynamic 55
3.3.3 Partial observation of CPN fault model 56
3.4 Related works 58

4 CPN diagnosis based on inequations system **60**
4.1 Diagnosis problem 60
4.2 Diagnosis of CPN by inequations system solving 64
4.2.1 Inequations system 65
4.2.2 Algorithms 67
$getImpossibleSols$ function 68
Diagnosis inferring 71
Multiple faults diagnosis 73
4.3 The minimality of CPN diagnosis 74
4.4 Related work 76

II Application **79**

5 Web services Application **81**
5.1 Introduction 81
5.2 SOA and Web service 81
5.2.1 SOAP 83
5.2.2 UDDI 84
5.2.3 WSDL 85
5.3 BPEL services 86
5.3.1 BPEL 86
5.3.2 Cooperation of BPEL and WSDL 87

5.3.3 ActiveBPEL engine . . . 88
5.4 Case study: foodshop . . . 88
5.4.1 Partners interactions . . . 89
5.4.2 BPEL services execution processes . . . 91
Customer . . . 91
Shop service . . . 91
Realsupplier service . . . 93
Warehouse service and LocalSupplier service . . . 94
5.5 Translate from BPEL to CPN . . . 94
5.5.1 Translating static BPEL features to CPNs . . . 95
5.5.2 Translation from basic Web service to CPN . . . 96
Basic Web service . . . 96
Receive(m, X) . . . 96
Invoke(X, Y) . . . 98
Reply(Y, m) . . . 100
Expression(C, V) . . . 101
Assign(X, Y) . . . 101
Throw/Rethrow$(faultName, [faultVariable])$ . . . 102
Wait$(duration|until)$ . . . 103
Empty . . . 104
Exit . . . 105
5.5.3 Structured operators translation . . . 105
Sequence operator $sequence(N_1, N_2)$ . . . 105
Conditional operator $Switch(\{(con_i(X_i, V_i), N_i)\}_{i\in I})$ . . . 106
Iterative operator $while(con(X, V), S_1)$ . . . 106
Message triggering operator $Pick(\{M_i, S_i\}_{i\in I})$ . . . 107
Parallel operator $flow(\{S_i\}_{i\in I})$ . . . 107
Conditional operator $If(\{(con_i(\overline{X_i}, \overline{V_i}), S_i)\}_{i\in I})$ . . . 107
Conditional operator $RepeatUntil(\{(con_i(X_i, \overline{V_i}), S_i)\}_{i\in I})$ . . . 109
5.5.4 sub process with enclosed environment: Scope . . . 109
Fault handlers . . . 111
$EventHandlers$ . . . 112
$CompensationHandler$ and $TerminationHandler$ . . . 112

Conditional operator $ForEach(\{(con_i(X_i, V_i), S_i)\}_{i \in I})$ 114
5.6 Case study: the CPN model of foodshop 114
5.7 Related works 115

6 Decentralized architecture for CPN based diagnosis 121
6.1 Introduction 121
6.2 Decentralized system 122
6.3 Diagnosis problem of decentralized system 123
6.4 Diagnosis approach 125
6.4.1 Diagnosis protocol 125
6.4.2 Diagnosis algorithm 128
6.4.3 Example: dining philosophers 131
6.5 Proof of global consistency of decentralized diagnosis 134
6.5.1 Functional CPN definition 134
6.5.2 Fundamental equations of functional subnets 135
6.6 Decentralized diagnosis of orchestrated BPEL services 138
6.7 Case study:foodshop 140
6.7.1 Exceptions 140
CUSTomer exceptions: 140
SHOP exceptions: 141
SUPPLIER exceptions: 141
WAREHOUSE exceptions: 141
6.7.2 Fault scenarios 142
6.7.3 Diagnosis 145

7 Conclusion 151
7.1 Future work 153

Bibliography 2

A The foodshop file list 2

List of Tables

3.1 Pre and $Post$ matrixes 45
3.2 Dining philosophers: incidence matrix 48
3.3 Dining philosophers: a characteristic vector 49
3.4 Dining philosophers: a characteristic vector of a transition sequence 49
3.5 Dining philosophers: incidence equation 50
3.6 Dining philosophers: incidence matrix with abstract data dependency 54
3.7 I/O data dependency of m_t 55
3.8 Dining philosophers: the incidence matrix 57
3.9 Partial observation example 57

4.1 Dining philosophers: an initial marking M_0 63
4.2 Dining philosopher: a symptom marking $\hat{M}$ 63
4.3 Dining philosophers: a characteristic vector $\overrightarrow{\delta}$ 63
4.4 Dining philosophers: a transition characteristic vector $\overrightarrow{\delta}_T$ 64
4.5 Dining philosopher: inequations system in form of matrix calculation 66
4.6 Dining philosopher: symptom marking $\hat{M}$ with multiple faults 74
4.7 Incidence matrix of the example illustrated in figure 4.4 75
4.8 Initial, symptom marking and characteristic vector of the example in figure 4.4 . . 75

5.1 Pre and $Post$ tables of a basic Web service 97
5.2 Pre and $Post$ tables of an asynchronous $Invoke$ activity 99
5.3 Pre and $Post$ tables of a synchronous $Invoke$ activity 99
5.4 Pre and $Post$ tables of $Assign$ activity 102
5.5 Pre and $Post$ tables of $Throw$ activity 103
5.6 Pre and $Post$ tables of $Wait$ activity 104

5.7 Pre and $Post$ tables of $Throw$ activity 104
5.8 Pre and $Post$ tables of $Exit$ activity 105

6.1 Dining philosopher: inequations system in form of matrix calculation on S_3 133
6.2 Dining philosopher: inequations system in form of matrix calculation on S_1 133
6.3 Dining philosopher: inequations system in form of matrix calculation on S_2 133
6.4 The activity names abbreviation of $Shop$ service 145
6.5 The activity names abbreviation of $RealSupplier$ service 145
6.6 The activity names abbreviation of $Warehouse$ service 146
6.7 The activity names abbreviation of $LocalSupplier$ service 146
6.8 The communication messages shared by the partners 146
6.9 foodshop example: decentralized diagnosis process 147

List of Figures

2.1 Basic perspective of MBD 13
2.2 A three dining philosophers problem 15
2.3 Dining philosophers: LTS of philosopher PH_1 20
2.4 Dining philosophers: LTS of $fork_1$ 21
2.5 Dining philosophers: LTS of $fork_1 \times PH_1 \times fork_2 \times PH_2 \times fork_3 \times PH_3$ . . 22
2.6 PN model of Dining philosophers 23
2.7 Dining philosophers: diagnoser of PH_1. The initial state of PH_{1_δ} is s_0. 27
2.8 Dining philosophers: $\mathcal{U}_{\mathcal{N}} \times obs$ for diagnosis 30
2.9 Coordinated decentralized architecture of DES 33

3.1 A CPN graph example 45
3.2 Dining philosophers: CPN model 47
3.3 Dining philosophers: CPN with abstract data dependency relations 54
3.4 Dining philosophers: CPN model for diagnosis 56

4.1 A solution for a CPN diagnosis problem 65
4.2 Apply the diagnosis algorithm with dining philosophers example 70
4.3 Apply diagnosis algorithm on 3-d-p example 74
4.4 Minimal diagnosis example 75

5.1 Composite Web service structure 82
5.2 The relations between the roles and protocols 82
5.3 SOAP structure 83
5.4 WSDL document structure 85
5.5 Interactivities between BPEL and WSDL protocols 88

5.6 The partners of the foodshop example . . . 90
5.7 The workflow of customer . . . 92
5.8 basic place/transition representations . . . 95
5.9 CPNs of the basic activity . . . 97
5.10 Pre and $Post$ tables of a $Receive$ activity . . . 98
5.11 CPNs of the receive activity . . . 98
5.12 CPN model of invoke activity: the thick-line places represent the remote places . . . 100
5.13 Pre and $Post$ tables of a $Reply$ activity . . . 101
5.14 CPNs of the reply activity . . . 101
5.15 Pre and $Post$ tables of $Expression$ activity . . . 102
5.16 CPNs of the expression activity . . . 102
5.17 CPNs of the assign activity . . . 102
5.18 CPNs of the throw/Rethrow activity . . . 103
5.19 CPNs of the wait activity . . . 104
5.20 CPNs of the $Empty$ activity . . . 104
5.21 CPNs of the $Exit$ activity . . . 105
5.22 CPNs of the sequence activity . . . 106
5.23 Additional Pre_i and $Post_i$ tables . . . 107
5.24 CPNs of the switch activity . . . 107
5.25 CPNs of the While, Pick, Flow, and If activities . . . 108
5.26 CPNs of the $RepeatUntil$ operator . . . 109
5.27 CPNs of the $faultHandlers$ activity . . . 110
5.28 CPNs of the $eventHandler$ operator . . . 111
5.29 CPNs of the compensationHandler and TerminationHandler . . . 112
5.30 CPNs of the $scope$ operator . . . 113
5.31 CPN model of forEach activity: the thick-line places represent the remote places . 114
5.32 A small BPEL process LocalSupplier . . . 115
5.33 The CPNs model of LocalSupplier . . . 116
5.34 The CPNs model of SHOP process . . . 117
5.35 The CPNs model of WAREHOUSE process . . . 118
5.36 The CPNs model of the SUPPLIER process . . . 119

6.1 Dining philosopher: as three distributed parts S_1, S_2, S_3 . . . 122

6.2 Place-bordered CPN model in a decentralized architecture 123
6.3 Decentralized diagnosis architecture of place-bordered CPNs 124
6.4 The flowchart of the local diagnoser 126
6.5 The flowchart of the coordinator 127
6.6 Dining philosophers: decentralized diagnosis process 132
6.7 The partners of the foodshop example 148
6.8 *INCORRECT ASSEMBLE OF PARCEL* 149

List of Definitions

1 LTS . . . 15
2 LTS path . . . 15
3 Alive LTS . . . 16
4 Convergent LTS . . . 16
5 Synchronous product of LTS . . . 16
6 Petri net graph . . . 17
7 Marking . . . 17
8 Petri net system . . . 17
9 Enabled transition . . . 18
10 Firing . . . 18
11 Firing sequence . . . 18
12 Reachable set . . . 18
13 Characteristic vector . . . 18
14 Marking equation . . . 18
15 Marking graph . . . 18
16 Bounded PN . . . 18
17 LTS fault model . . . 19
18 PN fault model . . . 19
19 Observable sequence . . . 25
20 Diagnosis of DES . . . 25
21 Minimal Diagnosis . . . 25
22 Diagnoser . . . 26
23 Diagnosis of LTS diagnoser . . . 27
24 PN Homomorphism . . . 28
25 Occurrence net . . . 28

26 Cut 29
27 Configuration 29
28 Branching process 29
29 Unfolding 29
30 Diagnosis of PN unfolding 29
31 Multi-set 43
32 Multi-set expression 43
33 CPN graph 44
34 Well-formed CPN Graph 46
35 Incidence Matrix of CPN 46
36 Fault model of CPN 46
37 CPN marking 48
38 CPN system 48
39 CPN mode enabling rules 48
40 CPN mode firing rules 48
41 CPN mode sequence firing rules 49
42 CPN reachability 49
43 CPN characteristic vector 49
44 FW function 52
45 SRC function 52
46 EL function 52
47 CPN fault model graph 53
48 Partial order observation 57
49 Partial order observation specificity 57
50 Partial order observation union 58
51 Minimal partial ordered observation 58
52 Minimal partial order observation function 58
53 CPN symptom markings 61
54 CPN diagnosis problem 61
55 Covering relation 61
56 CPN diagnosis 62
57 CPN minimal diagnosis 62
58 Multi fault operator 73

59 CPN fault model for basic WS 96
60 CPN fault model for $Receive$ activity 97
61 CPN fault model for an asynchronous $Invoke$ activity 98
62 CPN fault model for a synchronous $Invoke$ activity 99
63 CPN fault model for $Reply$ activity 100
64 CPN fault model for expression 101
65 CPN fault model for $Assign$ 101
66 CPN fault model for $Throw$ 103
67 CPN fault model for $Wait$ 103
68 CPN fault model for $Empty$ 104
69 CPN fault model for $Exit$ 105
70 Bordered places set 122
71 CPN Partnership 123
72 Decentralized CPN diagnosis problem 125
73 Decentralized CPN diagnosis 125
74 Functional CPN 134
75 Subnet of CPN 134
76 Minimal subnet of CPN 134
77 Communicating functional subnet union 137

Chapter 1

Introduction

From the point view of the system theory, a system is a set of interacting or interdependent entities forming an integrated whole. Structure, behavior, and inter-connectivity are keywords to define a system. Nowadays we rely more and more on the large systems and Internet for the facilities of the human daily life. That is the reason artificial intelligence is more and more applied in the system construction and maintenance process. As a robust and reliable large system (e.g. plant system, power system, aerospace system, Web service application) should be fault tolerant, self-healing (which is able to automatically recover possible failures) systems have been studied quite a lot during the last decade. So diagnosis, as mandatory step of self-healing system design, has a significant importance.

As a system contains a set of components (sub-systems), the performance of the whole system relies on the performance of each individual component as well as on the quality of the interaction between the components, which can be abnormal. The abnormal behavior of a component of a system is in many cases due to the occurrence of a fault within the component itself or due to abnormal behavior that is propagated via the interactions between the components. There are normally three different stages of diagnosis: fault detection, discovering whether a fault occurred; fault isolation, what kind of fault occurred; and fault explanation, what is the cause of a fault. For the convenience and cost saving, diagnosis is done locally, and if necessary, the overall (global) diagnosis can be integrated based on the local diagnoses.

The existing works seldom clearly and directly illustrate the abnormal data when diagnosing the abnormal behaviors of the system. while the abnormal data is quite easy to be detected and isolated, especially in the software systems. In this thesis, we focus on a software system which is a set of interacting distributed Web services application. Each Web service has no knowledge

of the structures of its sibling Web services because of distributed privacy policies. We tackled the abnormal behavior of a software system caused by the abnormal input data, control and/or the abnormal behaviors of the system components, instead of the faults in the programming codes. We make an important assumption:

Assumption 1. *The diagnosability is ensured by the sufficient observations, which are the logs of the Web service execution engine.*

There are normally three categories of diagnosis for a system: traditional, model-based, machine-learning based. The traditional approach is mainly rule-based method, which replies on expert knowledge and historical data and is usually implemented as an expert system. The rule-based diagnosis usually works under the principle set in form of ”If symptom(s) then fault(s)”. But it cannot be practically applied to large and complex systems since the knowledge acquisition is difficult and time cost is expensive. The rules set grows exponentially along the system size. And as it cannot well handle the plant structure changes, it is hard to maintain.

While the model-based diagnosis (MBD) [109] approach can naturally overcome these drawbacks, which is why we choose the MBD approach in this thesis. We have a model that describes an abstraction of the structure and behavior of the system, which can be incomplete. Given observations of the system, the diagnostic system simulates the system using the model, and compares the observations actually made to the observations predicted by the simulation. The task of the diagnosis system is to detect, isolate and explain the fault by consistency-based or abductive reasoning. The overall system model is simply adjusted when its structure or behaviors change by adjusting the model of the modified components instead of updating a huge database of rules. We refer to [31] for an extensive classification of the model-based diagnosis approaches. More specifically, we chose Colored Petri net (CPN) [64] as a model which unifies the data, control and the communications between the system components.

Machine learning methods are data-based and used when no explicit behavioral model is available (black-box model). They require a lot of observations made on the system and thus cannot be implemented at design stage. These methods can also be used in the case-based reasoning framework or, together with model-based approaches, for grey-box models, applied to the diagnostic knowledge itself: the previous successful or failed diagnoses, together with the available domain data, are used to automatically and continually improve system performance. This can be a nice improvement approach of the diagnosis, but is not the research interest in the thesis.

1.1 Abnormal behavior of software system

A software system is a system based on software forming part of a computer system (a combination of hardware and software), by focusing on their major components. The system theory, in software engineering context, is often used to study the large and complex softwares, which focus on their major components and the interactions between each other. Examples of software systems include computer reservations systems, air traffic control softwares, military command and control systems, telecommunication networks, web browsers, content management systems, database management systems, expert systems, spreadsheets, theorem provers, window systems, word processors, etc.

As in other complex systems, two approaches can be adopted in analyzing the behavior of a software system: the continuous dynamic one, and the discrete dynamic one. The continuous dynamic system changes its states as the continuous time elapses, while the state change of a discrete event system is driven by the discrete event occurrence. A software system is naturally suitable to be analyzed in a discrete dynamic way. As the state of the system can be represented by the state of all the data used by the system, while the evolution of the data set is driven by the events or the pre-defined processes in the system.

1.1.1 Communication between the components

The nature of the software system structure within an organization has yielded a choice between control (*centralization*) or coordination (*decentralization*). A system that is partially centralized and partially decentralized is termed a *distributed* system, which is more representative in the real world.

As all the complex systems, we have plenty of reasons to design the software systems in a distributed environment (for robustness, for efficiency, for cost control, for privacy protection, for concurrency, for autonomy, etc), which means the components are distributed on different network hosts. So the data transmission protocols for software systems are usually network-based, such as, HTTP, FTP, Telnet, SSH, POP3, SMTP, IMAP, SOAP, PPP, etc.

Different from the centralized software system, distributed software systems expose their internal component communication on the network, which rises more challenges to face when modeling and diagnosing them. For example, from the point view of fault handling, how to diagnose the fault comes from other components, how to minimize the fault infection in case of malfunction, how to handle the local faults without the additional information from the other components, etc.

1.1.2 Data and activities flow

For a complex software system, both the data set which represents the system status and the system structure which defines the system status evolution can be complicated. The data set is defined as a structured object, and the system is organized by the small parts like components (or modules, structured processes, functions) within the structure as sequences, parallels, choices, etc. The data is independently encapsulated within the functional scope of these components. And the communications between them are performed by sending messages (called input/output), which are a sort of pre-defined data structures, too. When the components are located on different sites, the software systems become distributed ones.

Data transforming activities

Within each component, data transformation is achieved by performing the pre-defined structured activities. And the legal order of the activities is decided by the topological structure and the choice of the path within the structure, which we call control. So control can be viewed as the dynamic system information, which is another kind of data of the software system.

The components communicate with each other by sending/receiving data (messages) while control information is handled in two ways: synchronous or asynchronous. In the synchronous communication, the sender component sends a message to the receiver and waits until the receiver returns before continuing, during which the control is temporally blocked and the whole process might be stuck. In the asynchronous communication, after the sender component sends the messages, it continues to execute until the process needs the receiver's returning message. So during the asynchronous communication, the control is not blocked, which avoids the ”stuck” bottleneck.

Dependency of Data

The general purpose of a software system is data manipulation, which means to compute the new system status according to existing data, which we call the dependency of data. Once the existing system status (called variables) contains faults, the faults could spread to other variables or other components. How the faults spread depends on how the new variables are manipulated from the existing ones. So it is necessary to define the data dependency to specify the way of fault spreading.

Moreover, beside the system variables, there are dependencies between the controls of the components. And sometimes, the controls, especially the choices of the paths, rely on both the former control information and the current system status. As both control and system variables are data,

the data dependency between the control and system variables can be classified just as that between the system variables.

1.1.3 Faulty state

As an artificial object, the abnormal behavior (either visible or invisible) of a software system can be caused by the design flaws, abnormal hardware or software environment, faulty inputs, or misinterpretation among the components. Not all the abnormal behaviors can be ignored or immediately solved by the system. In worse cases, in a large and complex system which may have long-term life cycle, tiny abnormal behaviors can cause serious consequences very late.

In addition of the programming mistakes which can be detected and corrected at the compiling stage, there are two categories of runtime abnormal behaviors we can detect in a software system. The first category is *error*, an irrecoverable condition occurring at runtime, which depends dynamically on architecture, OS and server configuration (e.g. out of system memory). The only way to recover it is to modify the system structure, replace the faulty module(s), or change the runtime environment configuration. The judged or hypothesized cause of an error is a *fault*. A *failure* occurs when an error ”passes through” the system-user interface and affects the service delivered by the system which is detected as the second category *exception*, an error condition that changes the normal flow of control in a software system (e.g. attempting to read an unavailable file, divide by zero, etc.). A signal that indicates an error condition is named as an *alarm* (alert). Both exceptions and alarms are called *symptoms*. An exception can be recovered probably by first requesting the correct inputs and then rolling back or restarting the software system. We assume there is no clear distinction between the exception and error. A tiny mistake, which should be caught as an exception can easily cause serious irrecoverable error, so we don’t make difference between exception and error in this thesis, and we call both of them as exceptions.

From the point view of the discrete dynamics, the information of a software system which contains faults is a *faulty state*. Theoretically it is a complete set of the properties of a system, but in practice, not all the properties are significant for diagnosing the faults. So we study only the significant property set and we name one configuration of a significant property set as a faulty state, similarly we name an abnormal event which cause the exception(s) as a faulty event.

1.2 Problematic of MBD for software systems

Diagnosis as a term is used in medical domains for diseases recognition on the basis of physical symptoms. ”Technical diagnosis, as a knowledge domain in Computer Science, and in particular in Artificial Intelligence, started developing 30 years ago. At the beginning, the objects of diagnostic interests were only mechanical machinery and devices. This set has been successively completed with electric devices, electronic systems, complex technological devices and recently with manufacturing and chemical processes as well as control systems”. [1].

1.2.1 The choice of abstract model

For an abstract model for diagnosis, a good model should represent as much as possible the system properties related to the diagnosis but in a simple way. These properties related to MBD are:

- all the data and control information handled by the software system, which can be labeled as correct or faulty.
- all the discrete events, which can behave correctly and faulty. Considering the possible frontiers of the distributed components because of the physical or privacy limits, the discrete event systems should allow to be composed and decomposed without violating the diagnosis result.
- all the data and control manipulation and data dependencies which reflect which data or event(s) is responsible for the symptoms.
- all kind of symptoms which can be reported by the different components of the software system: the clients, the monitoring components, or the execution components.

So to find a model with rich capability of representation and powerful mathematical or logical properties is always an important task for MBD.

1.2.2 The observation for diagnosis

The observations, as the information collection of the system states, should contain the information such that:

- the standardized event description, or standardized state descriptions: the values of the system variables;
- when: the time stamps of the events or states;

- where: the location of the events or states;
- how: the pre and post conditions (system states) of the events, the order of the events.

This process of information collection is called monitoring. Fortunately, the software system is much more easier to be monitored than the mechanical systems as the system data access does not require sensors and the software systems have no memory limits of monitoring. For example, the execution engine of a Web service application logs the value of the input, output variables, if the activities defined in the Web service are successfully executed, the alarms and exceptions during the execution. So it is easy to achieve the all the information mentioned above.

Imperfectness of observations

The large software systems consist of the components from different sites and providers and thus observations are geographically distributed; getting observation may also require authorization. Another problem is the clock synchronization of the different sites. Thus even if the monitoring of the software systems can offer rich observation information, the observation quality should be questioned. In case of components with limited authorization access, the observation could be missing or limited. And for observations from different sites with unsynchronized clocks, the observation may be only *partially ordered*. In the software systems, whether the functions are executed can be observed but the execution modes (correctly or faulty) cannot be observed.

Exception assertion

The exception assertion in the software system can come form two parts: the system execution exceptions or the feedback from the users. The exceptions during the system execution are aroused when the pre or post conditions (input or output variables of the system) of the pre-defined activities are not satisfied. These violations concern the value and attributes of the variable data. From the aspect of the faulty data value, it could be like the input variables are not (correctly) initialized, and so the output are not successfully evaluated. From the aspect of the faulty data attributes, the input/output could arrive or be generated within the wrong delay, from the wrong cooperating components, etc. The feedback from the clients or any manual assertion can be a source of exception assertion, which means any violation of the anticipated system behaviors can generate a system exception.

As mentioned above in section 1.1, each component must be provided with a quality contract

that can be expressed as a set of constraints. Violating one of those constraints will result in triggering an alarm. The alarms will be redirected to the diagnosis component and/or to a database in order to be treated as effectively as possible. So we will have to elaborate high-level alarms (or symptoms) which can be processed by the diagnosis component.

These alarms can be classified according to their corresponding occurrence level. We can distinguish infrastructure and middleware alarms due to failures in the underlying infrastructure (hardware, network); communication alarms due to failures in component invocation and organization (they are mainly application-specific alarms) and application alarms due to data mismatches, actor faults, coordination failures.

A diagnosis component is triggered by one or several symptoms reported by its associated monitoring component to search backward the diagnosis based on the data and control information recorded by the monitoring logs.

1.2.3 Minimal diagnosis

Based on all the former limitation for diagnosing the software systems, the diagnosis could be not accurate enough to discard all the unreal reasons. The diagnosis can give a super set which includes all the possible reasons which satisfy the known diagnosis conditions: the observation and symptom description. A minimal diagnosis is a diagnosis set that can explain all the symptoms while any of its subsets cannot. Two pieces of parsimony criteria are inferred in this description: the relevancy (causal association among initial causes and symptoms) and irredundancy (no proper subset of initial causes is itself a minimal diagnosis). So in this scenario, to find a minimal explanation for a complicated symptom is a chief object.

In this thesis, we contributed to calculate the minimal diagnosis based on the partial ordered observations which is enough to explain the symptoms. But it does not means a superset of a minimal diagnosis is also a diagnosis.

1.3 Our contribution

Our contribution lies in two aspects: a compact and powerful abstract model based on Colored Petri Net (CPN) [64] for a software system and an effective algebraic diagnosis approach based on backward reasoning [4] approach. The application is taken from the European research project: WSDIAMOND.

1.3.1 Model construction

The CPN model combines the strength of Petri nets [99], the synchronization of concurrent processes, with that of the programming languages, data types definition, and data values manipulation. In this thesis, the CPN model is used to define a unified framework which combines the data and control flow for dealing with the normal and abnormal behaviors of the software system. The basic ideas of using CPN as fault model are as follows:

- using places to represent data (and control), using transitions to represent the activities, and using arcs expressions to represent the data dependencies;
- using the places types (colors sets) to represent the data status. In the Web service application, we limit the colors as just three ones: corrupted data, correct data, and status unknown status data. But theoretically, more colors are allowed in the CPN diagnosis framework.
- using transitions modes to represent the correct or corrupted activities;

Furthermore, the functional CPN subnets are defined to represent the distributed components of a software system for diagnosis.

1.3.2 Diagnosis approach

The aim of diagnosis is, based on the CPN models and the partially ordered observations, starting from the symptoms marking, to explain the symptoms with a minimum set of explanations. That is, to find out the transitions behaved in faulty mode and the possible faulty tokens in the initial marking, which means the faults from outside of the subsystem,. Our diagnosis approach has two following advantages:

- profiting from the CPN mathematical properties to diagnose the symptoms by solving the symbolic inequations systems;
- defining a method that generates semi-automatically a CPN fault model directly from the CPN model of a system;
- allowing the observation of the system to be partially ordered without decreasing the diagnosis precision.

1.3.3 Decentralized topology

In this thesis, the application is a set of interacting WS-BPEL(Web Service Business Process Execution Language, see [90]) services located on different sites. WS-BPEL is used as the starting point to generate a model of the services that represents, at a sufficient level of granularity, the data dependencies between activities (processing and control activities) within one service and also between the involved partners services. Finally a *decentralized* diagnosis algorithm, based on the exploitation of the data dependencies, is proposed for the explanation process. Within this approach, some assumptions are made that can be considered as realistic in the present development of the Web services framework. The fault detection is based on exceptions, which means that one of the partners services will notify the dysfunction and start the diagnosis, and in addition the exception mechanism of WS-BPEL is enough rich to support detailed description of the dysfunction so as to provide a good starting point for the diagnostic process. The existing WS-BPEL execution platforms, i.e., the Active WS-BPEL platform used in this project, offer the possibility to log data at each step of the instance execution. To finish, the proposed diagnosis algorithm is decentralized by considering local diagnosers, one diagnoser for each partner service, dialoguing with one supervisor, or, more generally, a hierarchy of supervisors, which is a suitable architecture in the case of orchestrated Web services.

Therefore a decentralized diagnosis architecture is proposed which allows backward inference among the local CPN models which communicate through the bordered places. The equivalence between the global centralized diagnosis and the decentralized diagnosis is proved.

1.3.4 Application: WSDIAMOND project [138]

WSDIAMOND, Web Services - DIAgnosability[1], MONitoring and Diagnosis is a project of the Sixth Framework Program Priority2 - Information Society Technologies. It's a Specific Targeted Research or Innovation Project.

The possibility of creating self-healing software in order to guarantee reliability and availability of software systems and services is one of the main challenges for research The WSDIAMOND project is a first step towards self-healing software and specifically self-healing Web Services.

A self-healing Web Service is able to monitor itself, to diagnose the causes of a failure and to

[1]Diagnosability analysis is a part of the design stage and is based on models of the system. Diagnosability analysis provides information about the classes of faulty behavior of the system that can be diagnosed, which is a mandatory step in self-healing system design. Diagnosability can be evaluated at design time and may be involved in the software validation criteria.

recover from it, where a failure can be either functional, such as the inability to provide a given service, or non-functional, such as a loss of service quality. Self-healing can be performed at the level of a single service, and at a more global level, with support to identify critical misbehavior of groups of services and to provide Web Services with reaction mechanism to global level failures. The focus of WSDIAMOND is on composite and conversationally complex Web Services, where composite means that a Web Service relies on the integration of various other services, while conversationally complex means that during service provision a Web Service needs to carry out a complex interaction with the consumer application, where several conversational turns are exchanged between them.

In the project we tackled two main issues:

- On-line support: developing an operational framework for self-healing service execution of conversationally complex Web Services, where monitoring, detection and diagnosis of anomalous situations, due to functional or non-functional errors, is carried on and repair/reconfiguration is performed, thus guaranteeing reliability and availability of Web Services;
- Off-line analysis: designing a methodology and tools for service design that guarantee effective and efficient diagnosability/repairability during execution;

In order to achieve these goals, it carries on research in different areas such as Semantic Web languages (for describing service properties, i.e., models), service composition techniques (for describing service interaction, i.e. WS-BPEL) as well as model-based reasoning and diagnosis. This thesis is an extended work of the diagnosis part of the project and the CPN model and the diagnosis approach are applied on study case of the WSDIAMOND project.

The report is organized as follows: chapter 2 is a state of the art of the Model-based diagnosis of discrete event systems, which introduce the main modeling and diagnosis approaches; chapters 3 and 4 define our CPN model and diagnosis approach; chapter 5 concerns the application and adapts the CPN model for the interacting Web services; chapter 6 extends the diagnosis approach into a distributed system architecture and proposes a decentralized diagnosis protocol; chapter 7 is the conclusion and perspective of the thesis.

Chapter 2

Model based diagnosis of discrete event systems

2.1 Introduction

In the real world, scientists and engineers always face a fact that an artificial system (even human being himself) can never be guaranteed to be malfunction free. In other words, there may be some FAULT in the system. So one of their tasks for facing this difficulty is to answer to three questions: did any fault occur (detection)? if any, how many, where, and what kind of fault it is (isolation/identification)? and why and how did the fault(s) happen (explanation)? We call the answers to these questions as *fault diagnosis*.

Because of the growth of the complexity of artificial automatic systems, which is especially stimulated by the development of computer and Internet, fault diagnosis approach becomes more and more complex. Nevertheless the manmade nature of the artificial systems, it should be easy to get the explicit model of the system configuration and behavior to guide a diagnostic inference. So in this thesis, we follow the direction of model basic diagnosis (MBD) which is well-developed and quite mature for industrial application.

2.1.1 Model based diagnosis

As an application of abductive reasoning [94], the research on model-based diagnosis developed since the mid 70's [65, 30, 40] and led to several new methodologies, solutions and applications,

mainly applied to static systems. Those systems, with a unique non-changing state, supposed instantaneous observations, fault effects visible in the diagnosis window and no evolution of the system in this window. Associated methods resulted in timeless descriptions of the systems. Figure 2.1 illustrates the basic perspective of MBD.

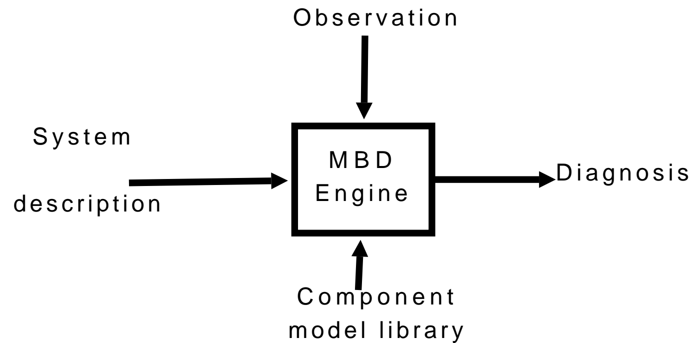


Figure 2.1: Basic perspective of MBD

Two distinct and parallel research communities have been working independently in MBD field: the FDI community and the DX community that have evolved respectively in the fields of Automatic control and statistical decision theory, and artificial intelligence and computer science. Readers are referred to [25] for more details.

2.1.2 Discrete event system

Systems can be modeled as continuous, discrete, or hybrid according to their dynamics of variables along the time line. Naturally there are two parallel research directions according to modeled systems: Continuous Systems (CS), modeled as algebro-differential equations or qualitative abstractions, and Discrete Event Systems (DES), modeled as finite state formalisms [128]. Some work ([136, 66, 53, 88, 11]) concerns the hybrid system as the system model, In this thesis, we model the software system as a DES.

”A Discrete Event System (DES) is defined as a dynamic system that evolves in accordance with abrupt occurrences, at possibly unknown, irregular intervals, of physical events” [106]. The model-based diagnosis of DES has received a lot of consideration over the past few years being applied in various technological areas. Besides the ”naturally discrete” systems (e.g. software system [142, 5, 20, 78]), the quantization of the variables’ change of the continuous ([116, 91, 83, 29]) and hybrid (like telecommunication networks [98, 97, 12, 37], power systems [8, 47, 56], plant systems [58], and production systems [14, 7, 146]) systems makes the discrete modeling possible.

2.1.3 Dining philosophers example

To illustrate and compare the different approaches of DES diagnosis, we use the example of three dining philosophers: N philosophers (ranging over $1 \cdots n$) sit around a table, one plate with one fork in it is placed between each two philosophers. So there are N plates (ranging over $1 \cdots n$) and N forks (ranging over $1 \cdots n$) on the round table. We consider a safe version of the problem where each philosopher takes and then releases both forks at the same time. In order to introduce a diagnosis problem, we consider two types of philosophers: well-organized and unorganized ones. A well-organized philosopher stops thinking when eating, so before he/she eats, he/she takes the forks and verifies whether the series numbers of the forks correspond to the series numbers of the plates at his/her left and right sides, if not, he/she will not eat. An unorganized philosopher never stops thinking. So he/she takes the forks on both sides and *never* verifies serial numbers. He/she mixes eating and thinking, so he/she might exchange the forks in his/her left and right hand when he/she crosses his/her hands during thinking. And this ”exchange” action is *unobservable*. Then he/she releases the forks to the wrong plates without conscience. Here we consider a 3-dining-philosophers problem (abbreviated as 3-d-p, see figure 2.2) where philosophers 1 and 2 are unorganized.

In the example, the activities of each philosopher taking and putting forks are observable, while the activity exchanging is unobservable. The series number of the forks are not observable except after the Philosopher 3 takes the forks and finds the forks’ series numbers are wrong.

In figure 2.2, the circles and the rectangles represent respectively the plates and the actions (take t_i, and release r_i with $1 \leq i \leq 3$). Note that ”exchange” action is not observable and ”eat” action is omitted as it has no relation to the diagnosis problem. The arrows linking the plates and actions represent the corresponding relations between them. The variable x_i (x'_i) labeled on the arrows represents the series number of the fork in the plate i taken (released) by the philosophers. The variables x_{ii} and x_{ij} (with $1 \leq i \leq 3$, $1 \leq j \leq 3$, and $j \neq i$) represent the fork series numbers in the right and left hand of philosopher i after ”taking” and before ”releasing”. So the sensors are placed on the actions t_i and r_i and the variables x_i, x'_i, x_{ii}, and x_{ij} which concern the series numbers of the forks during the system execution.

2.2 DES models

In this section, different DES are defined formally. First some common terms are given as follows: let L be a *set*, L^* denotes the *finite sequences set* over L, ε denotes the *empty sequence*, L^ϖ represents the *infinite sequences set* over L. $L^+ = L^* \backslash \varepsilon$ represents the *nonempty sequences set*. For

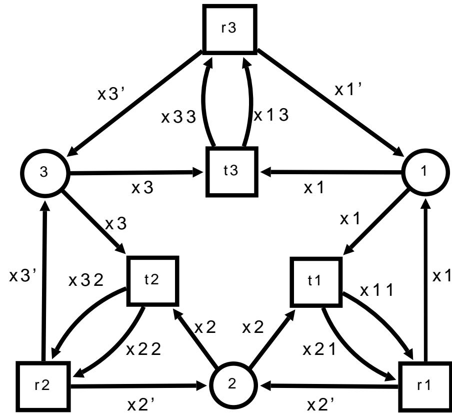


Figure 2.2: A three dining philosophers problem

sequences $\sigma, \rho \in L^*$, σ is a *prefix* of ρ, denoted as $\sigma \sqsubseteq \rho$ iff $\rho = \sigma\sigma'$, for some $\sigma' \in L^*$. And thus σ' is a *suffix* of ρ.

2.2.1 Labeled transition system and automata

When we are interested in studying a system dynamics which has a finite states and events set, a finite Labeled Transition System (LTS) [63] (or Automata [6] when final states set is distinguished), which is a directed graph with labels on the edges, is intuitively preferred, especially when the state space is small.

Definition 1 (LTS)**.** *A* labeled transition system A *is a tuple* $\langle Q, q_0, L, T \rangle$*:*

- Q *is a set of states,*
- q_0 *is the initial state,*
- L *is a finite set of events, with* $L = L_o \cup L_{uo}$ *and* $L_o \cap L_{uo} = \emptyset$*.* L_o *and* L_{uo} *are the observable and unobservable event sets.*
- $T \subseteq Q \times L \times Q$ *is a finite transition relation set.*

Definition 2 (LTS path)**.** *Let* A *be a LTS, then*

- *a* path *(or trajectory) in* A *is a finite or infinite sequence* $\delta = q_1 a_1 q_2 \cdots q_n$ *s.t.* $\forall 1 \leqslant i$*,* $(q_i, a_i, q_{i+1}) \in T$*. Denote by* $paths(q)$ *the set of all paths that start from the state* $q \in Q$ *and by* $paths(A)$ *the set of all paths in* A*, i.e.* $paths(A) = paths(q_1)$*. Denote* $q \in \delta$ *to represent*

the state q is in the sequence $q_1 a_1 q_2 \cdots q_n$ and $a \in \delta$ to represent that the event a is in the sequence $q_1 a_1 q_2 \cdots q_n$. Moreover, denote by $last(\delta) = q_n$ the last state of δ and by $|\delta| = n$ the amount of states in δ.

- *A* trace *σ of a path δ, denoted $trace(\delta)$, is the sequence $\sigma = a_1 a_2 \cdots a_n$ of events in L occurring in δ, and denoted $trace_o(\delta)$ is the sequence of the observable events in L occurring in δ. Write $traces(A) = \{trace(\delta) | \delta \in paths(A)\}$ for the set of all traces in A, particularly denote $traces^{\infty}(A)$ as the set of infinite traces of A. Write $traces_o(A) = \{trace_o(\delta) | \delta \in paths(A)\}$ for the set of all the observable traces in A. Moreover, in case σ is finite, let $|\sigma|$ denote the number of events occurring in the trace σ, i.e. $|\sigma| = n$.*

- *Write $q \xrightarrow{\sigma} q'$ if the state q' can be reached from state q via the trace σ, i.e. if there is a path $\delta \in paths(q)$ s.t. $last(\delta) = \sigma$ and $trace(\delta) = \sigma$. We write $q \rightarrow q'$, if there exists a trace σ s.t. $q \xrightarrow{\sigma} q'$, $q \rightarrow$, if there exists a state q' s.t. $q \rightarrow q'$, and $q \nrightarrow$ if q is the final state of σ.*

- *Given any trace $\sigma \in traces(A)$, denote by $\hat{\sigma}$ its* prefix-closure*, i.e. $\hat{\sigma} = \{\rho \in traces(A) | \rho \sqsubseteq \sigma\}$ and by $\check{\sigma}$ its* postlanguage*, i.e. $\check{\sigma} = \{\rho \in traces(A) \mid \sigma \sqsubseteq \rho\}$. Moreover, for a given natural number $k \in \mathbb{N}$, denote by $\check{\sigma}^k$ its postlanguage with only words with length longer than k, i.e. $\check{\sigma}^k = \{\rho \in \check{\sigma} \mid |\sigma| + k \leqslant |\rho|\}$.*

Definition 3 (Alive LTS)**.** *A LTS is* alive *iff $\forall q \in Q, \exists a \in L, q' \in Q$, s.t. $q \xrightarrow{a} q'$, i.e. iff $q \rightarrow$.*

Definition 4 (Convergent LTS)**.** *A LTS is* convergent *iff $\nexists q \in Q : q \xrightarrow{\delta} q$ and $\delta \in L_{uo}^*$.*

In this thesis, only the alive and convergent LTS are considered.

Definition 5 (Synchronous product of LTS)**.** *The* synchronous product *(or concurrent composition) $A_1 \times A_2$ (or $A_1 \| A_2$) of two labeled transition systems products a LTS $\Gamma = \langle Q_1 \times Q_2, L_1 \cup L_2, q_{0_1} \times q_{0_2}, T\rangle$, where T is defined as $((q_1, q_2), l, (q_1', q_2')) \in T$ iff :*

- $\forall l \in L_1 \cap L_2 \wedge (q_1, l, q_1') \in T_1 \wedge (q_2, l, q_2') \in T_2$,

- $\forall l \in L_1 - L_1 \cap L_2 \wedge (q_1, l_1, q_1') \in T_1 \wedge \nexists q_2' \in Q_2$, *s.t.* $(q_2, l, q_2') \in T_2$,

- $\forall l \in L_2 - L_1 \cap L_2$, $(q_2, l_2, q_2') \in T_2 \wedge \nexists q_1' \in Q_1$, *s.t.* $(q_1, l, q_1') \in T_1$.

The LTS example of the dining philosophers (presented in chapter 1 section 2.1.3) is given in figures 2.5.

2.2.2 Petri nets

Petri nets offer a graphical notation for stepwise processes that include choice, iteration, and concurrent execution. Petri nets have an exact mathematical definition of their execution semantics, with a well-developed mathematical theory for process analysis.

Definition 6 (Petri net graph)**.** *A* Petri Net graph *[99]* $(P/TNet)$ *is a triple* $\mathcal{N} = \langle P, T, W \rangle$*, where*

- P *is a finite set of places;*
- $T = T_o \cup T_{uo}$ *is a finite set of transitions,* $(P \cap T = \emptyset)$*. Let* T_o *is the observable transitions set,* T_{uo} *is the unobservable transitions set, and* $T_o \cap T_{uo} = \emptyset$*;*
- $W \subseteq (P \times T) \cup (T \times P)$ *is the incidence (flow) relation that specifies the arcs from places to transitions (Pre) and from transitions to places (Post):* $W = Pre \cup Post$*:* $Pre(p, t) : W_n(P \times T) \to \mathbb{N}$ *and* $Post(t, p) : W_n(T \times P) \to \mathbb{N}$*, where:*
  - $Pre(p, t) \in \mathbb{N}$ *gives the weight that is associated with the arc directed from place* p *to transition* t*;*
  - $Post(t, p) \in \mathbb{N}$ *gives the weight that is associated with the arc directed from transition* t *to place* p*;*

We represent by ${}^\bullet x = \{y \in P \cup T | (y, x) \in W\}$ and $x^\bullet = \{y \in P \cup T | (x, y) \in W\}$ respectively as the input and output places or transitions of x. The incidence relations Pre and $Post$ are in fact the $P \times T$ matrixes of $\mathbb{N}$.

To illustrate the dynamics of a Petri net, we define its execution with the notion of marking and a set of rules of marking evolution (transition firing). A marking is a distribution of tokens on places.

Definition 7 (Marking)**.** *A* marking M *of a net* $\mathcal{N}$ *is a* $|P|$*-vector that assigns to each place* p *of* P *a non-negative number of tokens* $M : P \to \mathbb{N}$*.*

$M(p)$ denotes the number of tokens present in p in the marking M. Markings are also called *configurations*. When we have a net with a marking, we have a net system.

Definition 8 (Petri net system)**.** *A* Petri net system $S\langle \mathcal{N}, M_0 \rangle$ *is a couple of a PN graph and an initial marking.*

The execution of a net system is based on the transition firing rules.

Definition 9 (Enabled transition)**.** *Let $S = \langle \mathcal{N}, M_0 \rangle$ be a Petri net system and M be a marking of S. We say that a transition t is* enabled *in M, if $\forall p \in^\bullet t$, $M(p) \geq Pre(p,t)$.*

When a transition is enabled, it can fire. The firing of a transition is the core concept of the execution of a Petri net.

Definition 10 (Firing)**.** *Let S be a Petri net system, M a marking of S, and t a transition enabled in M. The resulting marking M' of t* firing *from M (denoted as $M \xrightarrow{t} M'$), is defined as:*

$$M' = M - Pre(\cdot, t) + Post(t, \cdot)$$

by assuming $Pre(\cdot, t)$ and $Post(t, \cdot)$ are the $|P|$-vectors whose element p is $Pre(p,t)$, respectively $Post(t,p)$.

The definition 10 can be extended to a sequence of transitions:

Definition 11 (Firing sequence)**.** *A sequence of* firing transitions *is defined as a trace τ:*

$$\tau = M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{v-1}} M_v$$

where inductively for $i = 1, 2, \cdots, v-1$, $M_i \geq Pre(\cdot, t_i)$. So $M_1 \xrightarrow{\tau} M_v$ (named as legal trace*) denotes that τ fires at M_1 yielding M_v. The set of all legal traces in $\langle \mathcal{N}, M_0 \rangle$ is denoted by $\mathcal{L}_{\mathcal{N}}(M_0)$.*

Definition 12 (Reachable set)**.** *Given $S = \langle \mathcal{N}, M_0 \rangle$, the* reachable set *$\mathcal{R}_{\mathcal{N}}(M_0) = \{M | \exists \tau \in \mathcal{L}_{\mathcal{N}}(M_0) s.t. M_0 \xrightarrow{\tau} M\}$ is the set of all reachable markings.*

Definition 13 (Characteristic vector)**.** *The* characteristic vector *$\overrightarrow{\sigma}$ associated with a legal trace $\sigma \in \mathcal{L}_{\mathcal{N}}(M_0)$ is a T-vector whose element that corresponds with transition $t_i \in T$ is given by $\mu_\sigma(t_i)$ that is the number of appearances of t_i in the legal trace σ.*

Definition 14 (Marking equation)**.** *If $M_1 \xrightarrow{\sigma} M$, then the* marking equation *holds:*

$$M_1 + W \cdot \overrightarrow{\sigma} = M$$

(with the incidence relation $W = Post - Pre$ in a $P \times T$ matrix representation).

Definition 15 (Marking graph)**.** *Given a PN system $S = \langle \mathcal{N}, M_0 \rangle$, the marking graph of S is a LTS $MG = \langle \mathcal{R}_N(M_0), T, M_0, \Delta \rangle$ with $\Delta \subseteq \mathcal{R}_N(M_0) \times T \times \mathcal{R}_N(M_0)$ s.t. $\forall (M, t, M') \in \Delta$ iff t is enabled in M. So $M \geqslant Pre(\bullet, t)$ and $M' = M - Pre(\bullet, t) + Post(t, \bullet)$.*

Definition 16 (Bounded PN)**.** *$S = \langle \mathcal{N}, M_0 \rangle$ is* bounded *iff $\exists k \in \mathbb{N}$ s.t. $\forall p \in P$, $\forall M \in \mathcal{R}_N(M_0)$, $M(p) \leq k$. Then S is said to be k-bounded. If $k = 1$, S is said to be a 1-safe PN.*

2.3 Modeling diagnosis with DES

DES, as a dynamic system whose state changes with an event occurrence, can model the complicate man-made systems whose behaviors are hard to predict and offer the possibility to perform the automated fault diagnosis during the system execution. Commonly the faults are modeled as unobservable events and observation is modeled as a trace consisting of observable events. The DES diagnosis performs in two steps: deriving the legal traces which consistent with the observations; then make the assertion [115]:

- if all the traces include a same fault transitions, the fault is declared to have happened for *sure*;
- if none of the legal traces include a fault event, the diagnosis result is *normal*;
- if the legal traces set includes traces that include different fault transitions and/or do not include fault transitions, the diagnosis result is *uncertain*.

2.3.1 Fault representation

In some DES, faults set F can be alternatively represented as forbidden system states, faulty behavior modes of components, or unobservable events. In more complicated cases, unobservable events are categorized by *fault type*, which simplifies the fault representation. In this chapter, if not claimed in advance, each fault label has a different type, so F is a subset of events set. And N represents the normal state of DES.

Definition 17 (LTS fault model)**.** *A LTS $A = \langle Q, q_0, L, T \rangle$ is a fault model if a set of fault events are distinguished from the unobservable events set: $\exists F \subseteq L_{uo}$. The faulty events are ranged over by using f_i.*

Definition 18 (PN fault model)**.** *A Petri net $\mathcal{N} = \langle P, T, W \rangle$ is a fault model if a set of fault events are distinguished from the unobservable events set: $\exists F \subseteq T_{uo}$. The faulty events are ranged over by using f_i.*

3-d-p example 1. *The dining philosopher example consists of 6 modules: the philosophers ph_1, ph_2, ph_3; and the forks $fork_1$, $fork_2$, $fork_3$. As LTS does not support the real concurrent activity execution, each philosopher can perform the activities: take/put the left/right fork, or exchange (the forks in both hands). So there are 6 different pairs of possible series numbers of forks in the left or*

right hands of each philosopher in case philosopher 1 *and* 2 *are unorganized: 1/2, 2/1, 1/3, 3/1, 2/3, 3/2.*

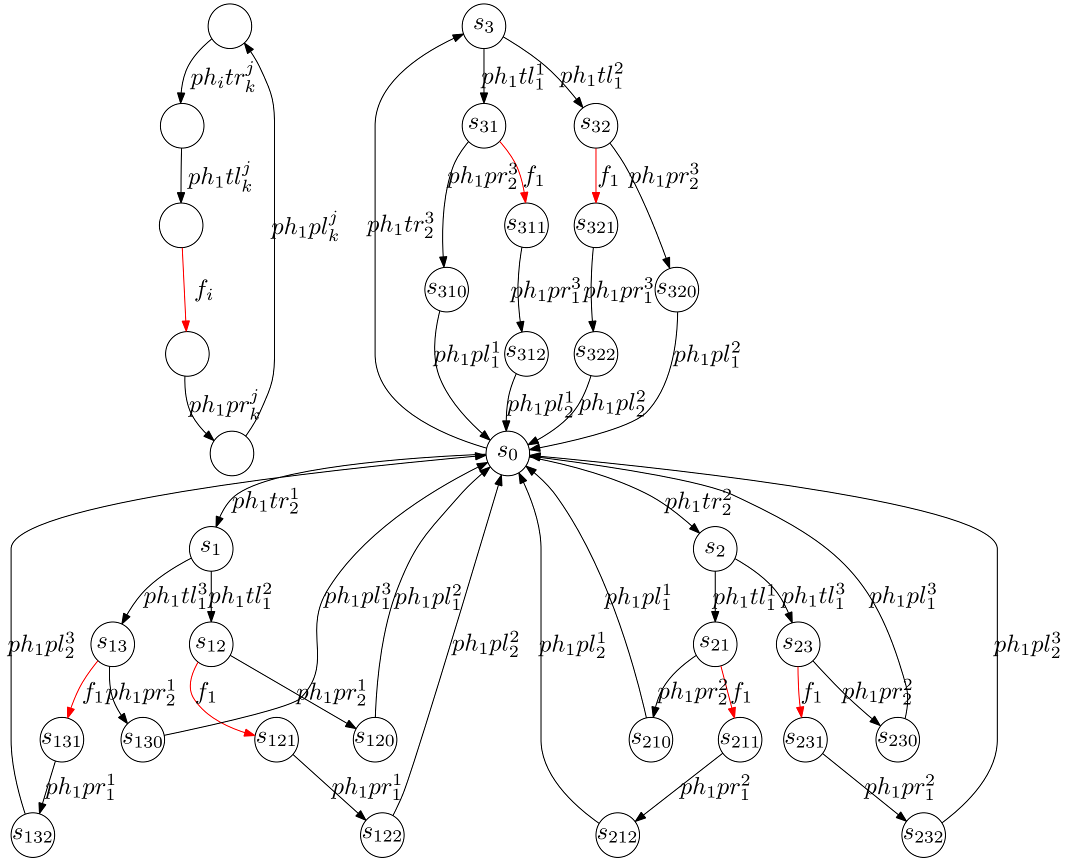
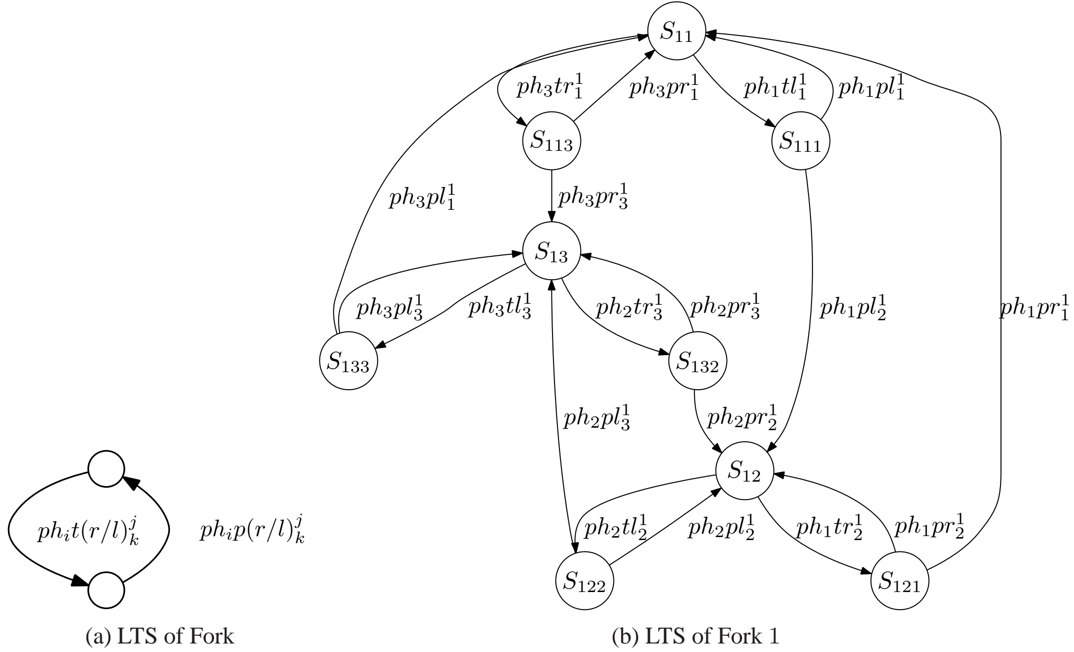


Figure 2.3: Dining philosophers: LTS of philosopher PH_1.

According to this different cases, the LTS (see figure 2.3) of each philosopher i *has 6 different groups of states. Each group starts from state* $S0$*, ends with* $S0$ *and performs in the activities order:* take right*(*$ph_i tr_k^j$*),* take left*(*$ph_i tl_k^j$*),* start to make mistake*(optional* f_i*),* put right*(*$ph_i pr_k^j$*), and* put left*(*$ph_i pl_k^j$*) fork* j *from/into plate* k *(see top left part of figure 2.3). For example, the activities chain,* S_0*,* S_2*,* S_{211}*,* S_{212} *and* S_0*, represents the philosopher* 1 *takes the fork* 2 *from the plate* 2 *with his right hand, takes the fork* 1 *from the plate* 1 *with his left hand, starts to make mistake, puts the fork* 2 *into his right hand in the plate* 1 *on his left side, then puts the fork* 1 *in his left hand into the plate* 2 *on his right side (see bottom left part of figure 2.3). For a state* s_i*,* i *represents the fork* i *in the left hand of philosopher. For a state* s_{ij}*,* i *represents the fork* i *in the right hand of philosopher and fork* j *in his/her left hand. For a state* s_{ijk}*,* i *and* j *have the same meaning with that of* s_{ij}*, and different*

values of k represents the different intermediate states. The initial state of LTS is s_0 which means the philosopher stands by.



(a) LTS of Fork

(b) LTS of Fork 1

Figure 2.4: Dining philosophers: LTS of $fork_1$.

Without considering the series numbers and left/right side, the LTS of the fork j is just a circle of being taken and put (see figure 2.4a). Considering the series numbers and left/right side, each fork may be placed in three different plates k ($k = 1, 2,$ or 3) and be taken or put by three philosophers i ($k = 1, 2,$ or 3). As the relative positions of the plates and philosophers are fixed, the LTS of fork j has three different groups of states which concern the three plates and three philosophers (see figure 2.4b). For a state s_{jk}, j represent the fork series number, k represents the plate series number. For example, the initial state of LTS is s_{11} which means the fork 1 is in the plate 1. For a state s_{jki}, i represents the series number of philosopher, which means the philosopher i has taken the fork j from the plate k. So a transition $s_{jk} \stackrel{ph_i tr^j_k}{\rightarrow} s_{jki}$ represents the philosopher i takes with his/her right hand the fork j from the plate k, like $s_{11} \stackrel{ph_3 tr^1_1}{\rightarrow} s_{113}$ and $s_{11} \stackrel{ph_1 tr^1_1}{\rightarrow} s_{111}$ (see figure 2.4b). The fork cannot make mistakes so, there is no f_i events in the LTS of the fork. But the three groups of states are connected by the activities when the philosopher 1 and 2 make mistakes.

So the synchronization product of $fork_i$ and PH_i ($i \in \{1, 2, 3\}$) $fork_1 \times PH_1 \times fork_2 \times PH_2 \times fork_3 \times PH_3$ is the LTS of the dining philosophers (see figure 2.5).

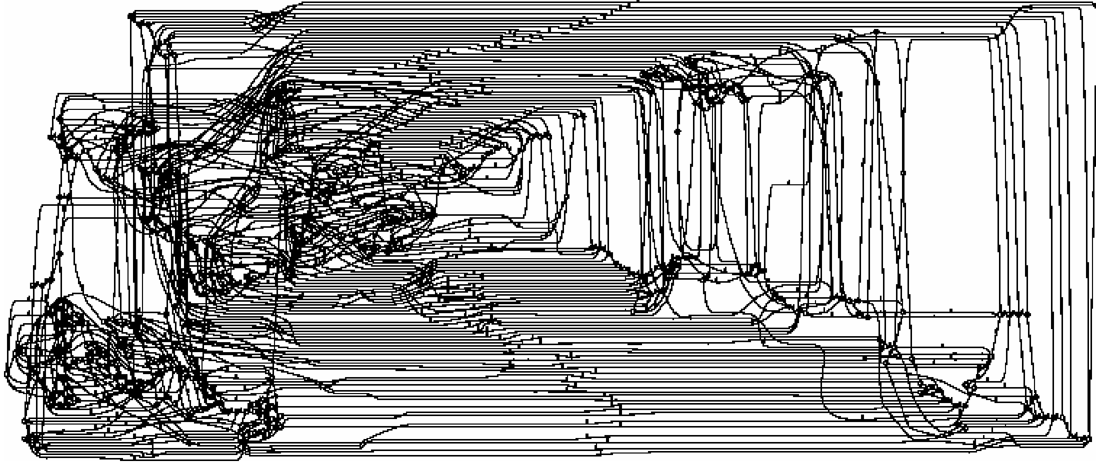


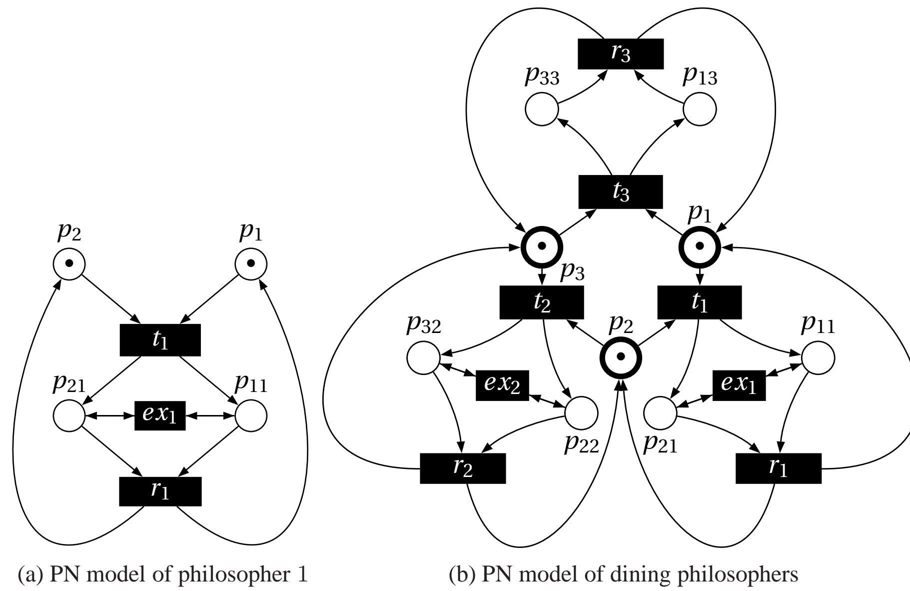
Figure 2.5: Dining philosophers: LTS of $fork_1 \times PH_1 \times fork_2 \times PH_2 \times fork_3 \times PH_3$ with 358 states and 58 events (with f_1, f_2, and f_3 three unobservable faulty events).

Note that there are some other kinds of fault representation like violation of event execution conditions [46], violation of constraints on the target states [93], or logical propositions defined over a set of variables that comprise both events and states [55].

3-d-p example 2. *We can model the dining philosophers example with PN model with the places represent the plates, the tokens in the places represent the forks (see figure 2.6a). Places p_1 and p_2 represent the plates 1 and 2 in each which have one fork. Places p_{11} and p_{21} represent the plate 1 and 2 without fork. Transitions t_1, ex_1 and r_1 represent the activities "take", "exchange" and "put" forks. As PN model allows the concurrent events, so taking, exchanging and putting forks happen concurrently. The other two philosophers 2 and 3 can model in the same way, and by composing the three PN models, we can get the PN model of the dining philosophers example (see figure 2.6b).*

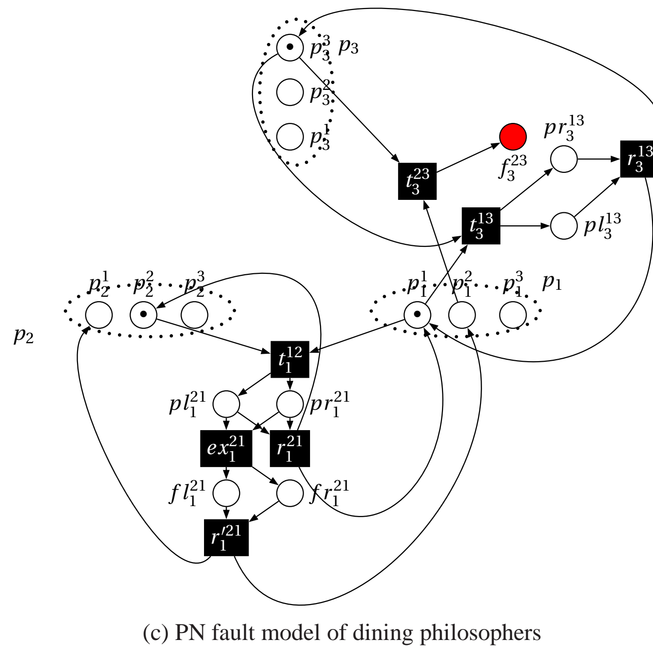
As addressed in example 1, the combination of series numbers of the forks in the left and right hands of the philosophers has 6 cases and these 6 kinds of forks combinations can be "taken" by either of the philosophers through activities t_1, t_2 and t_3. To totally isolate these cases in the fault model of PN, including the correct (3) and faulty (15) ones, there are 18 different kinds of "take" activities (transitions). Each "take" transition has its consequential "exchange" and "put" activities and corresponding places (see figure 2.6a).

Figure 2.6c illustrated the PN fault model of the dining philosophers example. Places p_k^j with $j, k \in \{1, 2, 3\}$ represent the cases, fork j in plate k. Transitions t_i^{jk}, r_i^{jk}, and ex_i^{jk} with $j, k, i \in$



(a) PN model of philosopher 1

(b) PN model of dining philosophers



(c) PN fault model of dining philosophers

Figure 2.6: PN model of Dining philosophers

$\{1, 2, 3\}$ and $j \neq k$ represent activities of philosopher i ”take”, ”put” and ”exchange” the forks j and k in his/her left and right hands. To make the figure visible, we only draw only 3 different cases t_1^{21} (philosopher 1 behaves well), t_3^{13} (philosopher 3 does not detect fault) and t_3^{23} (philosopher 3 detect the fault) and omit the other 15 cases. Note the red place f_3^{23} represents an alarm, which means philosopher finds the fork series number in his left hand is wrong. The dotted eclipses p_k circle all the possible series numbers of forks in plate k.

2.3.2 Observation

The observation of DES is retrieved from the monitoring system, which supervises the running of the system. Once it captures a symptom, the diagnosis process is triggered. The observation offered by the monitoring system is event-observations (as observation trace) and the partial state-observation (as symptom).

Assumption 2. *Unless otherwise stated, we make an important assumption:*

the fault cannot be in the monitoring components that log the information, which means the observations is accurate.

For DES, the most common observation is the occurrence of events. In reality, the sensors or monitoring platform in charge of the observations can be malfunctioning. So the observation sequence can be inaccurate, incomplete, partially ordered, etc. In fact, many works effort to completely or partially release these assumptions to meet the real-life request from the industrial areas.

Observation absence

Due to the limitation of the observation, there could be an observation absence, e.g., some states or events occurrence are naturally hard or too expensive to capture. Then the diagnosis problem is explored in two directions: to improve the diagnosis confidence with available observation, or to carefully configure the sensors with higher diagnosis confidence and lower cost.

As to the diagnosis confidence improvement, [113] studied the case of partial observation of system states and observable events in form of Petri nets. A (transition) labeled Petri net is defined. And a label function is defined to transform the combination of state observation and observable event into a transition label. And finally the diagnosis problem is defined as a PN reachability graph search with the help of stochastic information.

Partially ordered observation

An asynchronous system, much like an object-oriented software and a telecommunications network management system, is a system operating under distributed control, local time, global supervision, and components communication. Each local sensor has only a partial view of the system, and its local time is not synchronized with that of other sensors.

Even if the order of events may be correctly observed locally by each individual sensor, communicating alarm events via the network causes a loss of synchronization: as a result, the interleaving of events communicated to the supervisor is *nondeterministic*.

So we formally define the observation set OBS as follows:

Definition 19 (Observable sequence)**.** *Given an observable set L_o, a (partially ordered) observable sequence is defined as:*

$$obs ::= \varepsilon | event \prec obs | obs \parallel obs, \text{ with } event \in L_o$$

with ε represents the empty observation, $\prec$ and $\parallel$ represent respectively the precedent and parallel relations between the events.

2.3.3 Diagnosis of DES

So generally, the diagnosis of DES Δ_{DES} can be informally defined as follows:

Definition 20 (Diagnosis of DES)**.** *The diagnosis of a DES is a function $\Delta_{DES} : traces_o(A) \rightarrow 2^{2^{F \cup \{N\}}}$, with:*

- *$traces_o(A)$ is the set of observable traces;*
- *F is the set of fault types, N represents the normal state of the DES;*

Definition 21 (Minimal Diagnosis)**.** *A* minimal diagnosis *is a diagnosis Δ_{DES} such that $\forall \Delta'_{DES} \subset \Delta_{DES}$, Δ'_{DES} is not a diagnosis.*

Following the principle of parsimony, minimal diagnoses are often the preferred ones and proved, in particular cases (the weak fault model [32]), to be sufficient to characterize all the diagnoses. The proposition does not hold for the strong fault model[32], which is our case. In this thesis, each minimal diagnosis in the minimal diagnosis set is sufficient to explain the symptom.

2.4 Diagnosis methods

DES diagnosis methods are based on observing system events and making inferences about the system state. The basic idea is that the occurrence of a fault will generate a unique sequence of observable events that will establish the presence of the fault.

The classical diagnosis approach is to synchronize the system model for diagnosis and the observed traces for computing all compatible trajectories and determining whether these trajectories (a sequence of states and transitions) are normal. The system model for diagnosis can be represented as:

- synchronization product of DES model and fault types (diagnoser) [115, 116, 147, 129, 120] and PN diagnoser [8];
- PN unfolding [12, 37] and backward unfolding [58].
- Petri net reachability graph [4, 100, 3, 131, 46, 41, 17];

In this section, the above approaches are introduced and compared. There are other diagnosis approaches which are not widely used like consistency-based [49, 82, 126, 142, 141, 50, 144] and Algebraic approach based on Petri nets [76, 108, 72, 75, 73, 74, 110] which are summarized separately.

2.4.1 Diagnoser

Assume faults are represented as unobservable events and $F \subseteq L_{uo}$ is the fault types set (see definition 1). Given a LTS, the composition product of the system states and the possible faults types can represent off-line the diagnosis states of the system, named as diagnoser [115, 116]. So a diagnosis can be got by synchronizing the diagnoser and an observable trace.

Definition 22 (Diagnoser)**.** *Given a LTS $A = \langle Q, q_0, L, T \rangle$, a diagnoser based on A is a LTS $A_\delta = \langle Q_\delta, q_{0_\delta}, L_\delta, T_\delta \rangle$ with:*

- $Q_\delta \subseteq Q_o \times 2^{F \cup \{N\}}$ *with* $Q_o = \{q_o\} \cup \{q : \exists (q', a \in L_o, q) \in T\}$*;*
- $q_{0_\delta} = (q_0, \{N\})$ *is an initial state;*
- $L_\delta = L_o$ *is the set of observable events of* A*;*

- $T_\delta \subseteq Q_\delta \times L_\delta \times Q_\delta$ *is the finite transition set* $(q, F) \xrightarrow{a} (q', F')$ *s.t.* $q \xrightarrow{\sigma a} q'$ *with* $\sigma \in L_{uo}^*$, $a \in L_o, F' = F \cup \{f_i \in F | f_i \in \sigma\}$.

Definition 23 (Diagnosis of LTS diagnoser)**.** *The diagnosis of LTS diagnoser is an implementation of* Δ_{LTS} *on the DES* $A_\delta \parallel Traces_o(A)$. *Let* $\tau \in Traces_o(A)$, $\Delta_{LTS}(\tau) = \vee \phi_i$ *s.t.* $(q_\delta, \phi_i) \in (A_\delta \parallel \tau) \wedge (q_\delta, \phi_i) \nrightarrow$.

Diagnosis with diagnosers is very efficient because processing an observation sequence can be done in linear time in the length of the sequence. However, the construction of the diagnoser may be extremely expensive because the diagnoser may have a size that is exponential in the number of states in the system, which is famous as the *state-space explosion* problem.

3-d-p example 3. *Take the LTS of the philosopher 1* PH_1 *as example, its diagnoser* PH_{1_δ} *is illustrated as in figure 2.7.*

Figure 2.7: Dining philosophers: diagnoser of PH_1. The initial state of PH_{1_δ} is s_0.

As the number of the unobservable faulty events is far more less than that of the observable ones, it is obvious that the size of the diagnoser A_δ *is close to the size of the LTS of the dining philosophers (omitted).*

Given an observation obs $= ph_1tlp_1t_1, ph_1trp_2t_2, ph_1plp_2f_1, ph_1prp_1f_2, ph_2tlp_1t_2, ph_2trp_3t_3, ph_2plp_3f_2, ph_2prp_1f_3, ph_3tlp_1t_3, ph_3trp_2t_1$ *which is followed by* $Error$*, the diagnosis can be generated by simulating the diagnoser and the observation. So* $Diag_{\mathcal{U}_{PN}} = \{\{f_1, f_2\}\}$ *which means the only explanation of the symptom is both the first and second philosophers exchanges their forks.*

[115], [116] described a modeling and diagnosis framework for systems in the DES framework. A diagnoser based on the system model functions as an extended observer that provides estimates of the system state under non-faulty and faulty conditions.

[118] proposed a diagnoser approach by combining each marking with its exclusive diagnosis information, and got diagnosis by synchronizing the diagnoser with the observations. [118] requires the PN model to be more specific so that each marking corresponds either to a correct state or to one type of fault.

[103] improved the diagnoser approach of [115] by constructing *nondeterministic* diagnosers *off-line* and the *on-line* diagnosis is performed by maintaining a single *Reach* set and updating it upon each observation.

2.4.2 PN unfolding

Net unfolding [87] is a technique of structural analysis to reduce the state-space explosion problem which the reachability analysis approaches suffer from. The unfolding of a system fully describes its concurrent behavior in a single branching structure, representing all the possible computation steps and their mutual dependencies, as well as all reachable states; the effectiveness of the approach lies in the use of partially ordered runs, rather than interleavings, to store and handle explanations extracted from the system model.

The unfolding definitions are taken from [12] and slightly adjusted.

Definition 24 (PN Homomorphism)**.** *Given two PN graphs $S = \langle P, T, W\rangle$ and $S' = \langle P', T', W'\rangle$, a homomorphism from s to s' is defined as $\varphi : P \cup T \rightarrow P' \cup T'$ s.t.,*

- $\varphi(P) \subseteq P'$ *and* $\varphi(T) \subseteq T'$
- $\forall x \in P \cup T$, $\varphi(^{\bullet}x) =^{\bullet} \varphi(x)$ *and* $\varphi(x^{\bullet}) = \varphi(x)^{\bullet}$

Definition 25 (Occurrence net)**.** *Given a PN graph $S = \langle P, T, W\rangle$, two nodes x, x' are in* conflict*, noted as $x\sharp x'$, if $\exists t, t' \in T$, s.t. $^{\bullet}t \cap^{\bullet} t' \neq \emptyset$ and $t \preccurlyeq x$, $t' \preccurlyeq x'$ where $\preccurlyeq$ is a reflexive transitive closure of W. A node x is in* self-conflict *if $x\sharp x$. An occurrence net $\mathcal{O} = (B, E, W)$ satisfies:*

- *B, a set of conditions;*
- *E, a set of transitions;*
- *$\prec$ is the causality relation;*

- $\forall x \in B \cup E : \neg[x \sharp x]$ *(acyclic);*
- $\forall x \in B \cup E : \neg[x \prec x]$*;*
- $\forall x \in B \cup E : |\{y : y \prec x\}| < \infty$*;*
- $\forall b \in B : |{}^{\bullet}b| \leq 1$*, each place has at most one input transition (no backward conflict).*

We denote $min(\mathcal{O}) \subseteq B$ *as the minimal*[1] *node set of* $\mathcal{O}$ *for* W*.*

Definition 26 (Cut)**.** *Two nodes* x*,* x' *are concurrent, denoted as* $x \perp x'$ *if neither* $x \preccurlyeq x'$*, nor* $x' \preccurlyeq x$*, nor* $x \sharp x'$*. A maximum concurrent conditions (or pairwise nodes) set is a* cut*.*

Definition 27 (Configuration)**.** *A* configuration $\mathcal{C} = \langle B_{\mathcal{C}}, E_{\mathcal{C}}, \preccurlyeq_1\rangle$ *of* $\mathcal{O}$ *is defined as follows:*

- $\mathcal{C} \subseteq \mathcal{O}$*,* $\mathcal{C}$ *is a sub-net of* $\mathcal{O}$*;*
- $\forall a, b \in (B_{\mathcal{C}} \times E_{\mathcal{C}}) \cup (E_{\mathcal{C}} \times B_{\mathcal{C}}) \Rightarrow \neg(a \sharp b)$*,* $\mathcal{C}$ *is conflict-free;*
- $\forall b \in B_{\mathcal{C}} \cup E_{\mathcal{C}} : a \in B$ *and* $a \preccurlyeq_1 b \Rightarrow a \in B_{\mathcal{C}} \cup E_{\mathcal{C}}$*,* $\mathcal{C}$ *is up-warded closed;*
- $min_{\preccurlyeq}(\mathcal{C}) = min_{\preccurlyeq}(\mathcal{O})$*,* $\mathcal{C}$ *and* $\mathcal{O}$ *have the same starting nodes.*

We denote $\mathfrak{C}$ *as the configurations set of* $\mathcal{O}$*.*

Definition 28 (Branching process)**.** *Given a Petri net system* S*, a* branching process $\mathcal{B}$ *is a pair* $(\mathcal{O}, \varphi)$ *where* $\mathcal{O}$ *is an occurrence net and* φ *is a homomorphism from* $\mathcal{O}$ *to* S*, with:*

- $min(\mathcal{O}) = M_0 \Rightarrow \varphi(min(\mathcal{O})) = M_0$
- $\forall e, e' \in E$*,* ${}^{\bullet}e = {}^{\bullet}e' \wedge \varphi(e) = \varphi(e') \Rightarrow e = e'$

Definition 29 (Unfolding)**.** *Given a Petri net system* $S = \langle \mathcal{N}, M_0\rangle$*, the* unfolding $\mathcal{U}_{\mathcal{N}}(M_0)$ *is a branching process* $\mathcal{B} = (\mathcal{O}, \varphi)$ *s.t.* $\forall \mathcal{B}' = (\mathcal{O}', \varphi') \sqsubseteq \mathcal{B}$ *where* $\mathcal{B}'$ *is a prefix of* $\mathcal{B}$*,* $\exists$ *a homomorphism* $\phi : \mathcal{B}' \rightarrow \mathcal{B}$*, s.t.* $\phi(min(\mathcal{B}')) = min(\mathcal{B})$ *and* $\varphi \circ \phi = \varphi'$*.*

So $\mathcal{U}_{\mathcal{N}}(M_0)$ maximally unfolds S and configurations are the adequate representations of the firing sequences of S.

So the diagnosis based on Petri net unfolding can be defined as:

[1] $min_{\preccurlyeq}(X) = \{x \in X | (x' \in X \wedge x' \preccurlyeq x) \Rightarrow x' = x\}$ is the minimal element of X.

Definition 30 (Diagnosis of PN unfolding)**.** *Given a diagnosis problem $\langle \mathcal{U}_{\mathcal{N}}, obs \rangle$ with $\mathcal{U}_{\mathcal{N}}$ the unfolding of a Petri net system $\mathcal{S} = \langle \mathcal{N}, M_0 \rangle$, the diagnosis is $Diag_{\mathcal{U}_{P_{\mathcal{N}}}} = \{t_{uo} | t_{uo} \in T_{uo}, \exists \tau$ is an observable trace of $\mathcal{N}$, s.t. τ is consistent with $\mathcal{U}_{\mathcal{N}} \times obs\}$.*

3-d-p example 4. *Figure 2.8 illustrates a part of unfolding of 2.6c which is consistent with an observation $obs = t_1 r_1 t_2 r_2 t_3$ and symptom with place f_3^{21} is marked. So $Diag_{\mathcal{U}_{\mathcal{N}}} = \{\{ex_1, ex_2\}\}$ which means the only explanation of the symptom is both the first and second philosophers exchanges their forks.*

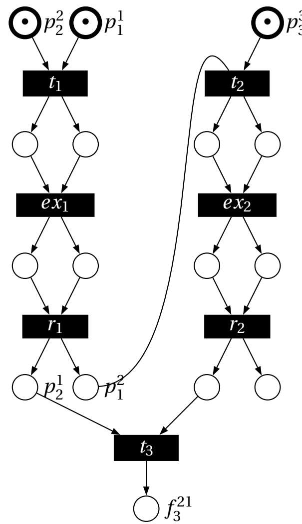


Figure 2.8: Dining philosophers: $\mathcal{U}_{\mathcal{N}} \times obs$ for diagnosis: the superscripts and subscripts of the intermediate places and the superscripts of the transitions are omitted.

[12] used a net unfolding approach for designing an on-line asynchronous diagnoser. The state explosion is avoided but the on-line computation can be high due to the on-line building of the PN structures by unfolding.

2.4.3 PN backward reachability analysis

Reachability analysis has been successively developed essentially by taking into account forward reachability. While backward reachability analysis is suitable for the diagnostic problem solving [4, 58, 121, 18]. The backward reachability analysis starts from the final marking which represents a symptom and calculates backwardly according to the backward searching rules to detect all the traces that cover it. So the backward calculation can be seen as a forward calculation in the reverse

PN obtained by reversing the direction of the arcs in the original PN and modifying the enabling and firing rule of a transition.

[4, 100] proposed the backward reachability analysis (*B-W analysis*) approach to model the behavior of a system to be diagnosed. The states of the system are represented as places, and the inferring relations between the states are represented as transitions. So the PN model represents all the possible logical inferring paths of the system states. The *B-W analysis* is to start from the final marking which is the observed symptom, and search backwardly all the consistent paths to decide the possible initial markings.

[58] adapted the PN unfolding method for backward searching. The set of minimal explanations is calculated backwards starting from the observation and deriving traces that lead back to the initial marking. The diagnoser explores different state spaces but has the advantage that it does not depend on the size of the PN model but only on the size of the largest sub-net in the model that includes only unobservable transitions. Moreover and very important the set of complete explanations can be calculated from the set of minimal explanations whenever this is required.

[41] studied the minimal diagnosis of unobservable-transitions-acyclic PN. A diagnosis approach named as *basic reachability tree* is proposed which is in fact an automaton [2] diagnosis based on marking graph of Petri net. [17] studied the reachability graph diagnosis approach based on bounded PN model. The observations are transferred to a *justification-vector* to improve the efficiency.

[131] introduced a method for modification of reachability trees in order to detect failure transitions. A symbol ϖ means an infinite set of positive integers, so an infinite tree consisting of infinite reachable markings is approximated by a finite tree (reachability tree). Two kinds of diagnosers (difference marking ϖ-diagnoser and refined ϖ- diagnoser) were proposed. For observable places whose token numbers are replaced by ϖ in the reachability trees, the former diagnoser calculates difference between token numbers before and after partially observed markings change, and detects failures. In the latter diagnoser is refined to distinguish the reachable markings by normal and faulty behaviors.

[46] used PN models to introduce redundancy into the system and additional P-invariants (places set whose tokens number produced/consumed by the I/O arcs are equal) allow the detection and isolation of faulty markings.

[2]The convergent (unobservable-transitions-acyclic) property makes sure the generated automaton is diagnosable.

2.5 Architecture of DES diagnosis

The large DES systems are usually designed as a set of interconnecting subsystems with different topological architecture, which can be roughly divided as decentralized and distributed ones. So decentralized and distributed diagnostic protocols become necessary to deal with diagnosis in distributed systems where the information is separately located.

The model-based diagnosis of DES can be classified in the literature from a topological point of view as centralized, decentralized, and distributed approaches.

2.5.1 Centralized diagnosis

There is one centralized diagnoser that derives the system diagnosis based on its (complete) knowledge of the overall system model and the overall system observation. The centralized approach can be further classified as:

- diagnoser approach [115] where a diagnoser automaton is derived off line and the on-line analysis is carried out by eliminating the diagnoser-states that are not consistent with the system observation.

- active system approach [8] where the diagnosis result is derived a posteriori when the system is in a quiescent state (out of work or idle).

The main disadvantage of a centralized approach is its high computational complexity. It requires a centralized model and generates a centralized diagnoser. Since the diagnoser-automaton can be viewed as a special observer-automaton its size may become too large to be practically stored [107]. Even if a centralized diagnoser can be constructed it has the following disadvantages [125]:

- weak robustness: when the centralized diagnoser is broke down, the whole system is not able to be diagnosed.

- low maintainability: a change in the system structure requires a complete re-calculation of a new centralized diagnoser, which can be a serious problem for the dynamic systems.

2.5.2 Decentralized diagnosis

The decentralized diagnosis problem is first considered in [33] in which the local diagnosers communicate with the coordinator through the no-delay channels in stead of with each other. Figure 2.9

illustrates the coordinated decentralized architecture with two local sites and communicating with a coordinator. There is one coordinating agent receives information from several local diagnosers, each of which performs some local diagnosis of the system with incomplete knowledge (e.g. based on a sub-set of sensor readings or a partial knowledge of the overall model). The local diagnosis results are compiled in a consistent diagnosis result for the overall system by the coordinating diagnoser e.g., [33, 97, 34, 98, 13].

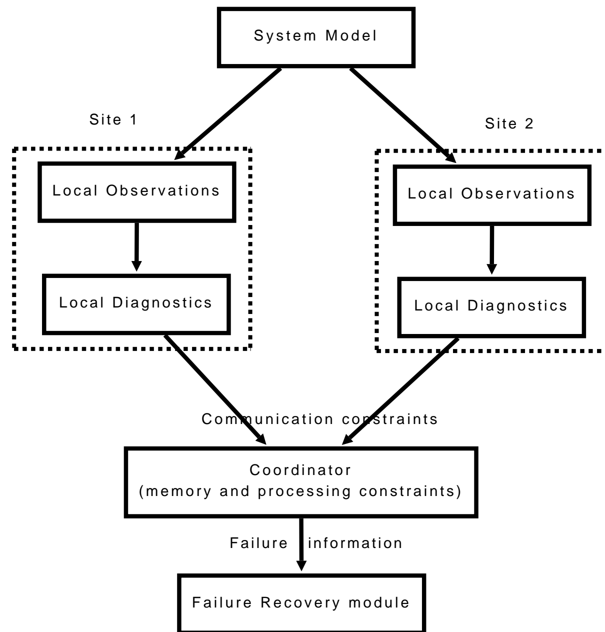


Figure 2.9: Coordinated decentralized architecture of DES

There are two different decentralization levels:

- [33], or its extended version [34, 120], employ a global system model which is built from component models automatically via synchronous or asynchronous composition. After off-line diagnosability verification, which may cause state explosion problem, the online diagnosis decisions can be computed. These decisions may or may not be fused on a coordinating site, according to the properties of the architecture. Three coordination protocols are proposed in [33] that realize the proposed architecture and analyze the diagnostic properties of these protocols.

- [8, 97, 71, 96] and [139] (synchronous automata) proposed the decentralized system model as asynchronous *communicating automata (or FSMs)*. [8] solved off-line a diagnosis problem *a posteriori*, while [71] mixed a diagnoser approach [120, 97] with an extended version of

the decentralized model of [8] by computing on-line only the interesting parts of a centralized diagnoser to avoid computing the global model. [96] introduced the temporal window to improve the on-line diagnosis efficiency and the global diagnosis is built by dynamically merging the local ones to eliminate the inconsistent traces with the partial order reduction technique and incremental diagnosis on sound temporal windows. Recent works on this approach [26] used decentralized or factored representations to represent the set of all trajectories more compactly without enumerating all of them.

While decentralized models could potentially reduce the state space exponentially, the actual complexity of the diagnosis algorithms relies on the partition of the system model and the selection of communicating events between local models.

[111] investigated the necessariness of asynchronous communication for fault diagnosis. It modeled the asynchronous communications between two local diagnosers with timed automata. Then the problem of determining the states of each of the two communicating diagnosers into the problems of factorization of the observation map and construction of an observer for a timed DES. The diagnosers can be formulated directly from the observers.

[13] studied the problem of synthesizing communication protocols and failure diagnosis algorithms for decentralized failure diagnosis of DES with costly communication between diagnosers. The costs on the communication channels may be described in terms of bits and complexity. The costs of communication and computation force the trade-off between the control objective of failure diagnosis and that of minimization of the costs of communication and computation.

[44] proposed a modular diagnosis architecture (broker) capable of merging diagnoses provided by local diagnosers and to enrich their formalism with synchronization constraints. The global diagnoser algorithm manages a diagnosis tree by querying the local diagnosers to complete the pending paths. Each candidate diagnosis is represented by a path leading to a constraintless node in the diagnosis tree.

The decentralized approaches overcome the high complexity and the low maintainability limitations of the centralized approach by calculating local state spaces (of size a lot smaller than the size of the overall) that are maintained consistent by a centralized structure (agent). But the existence of a centralized agent does not eliminate the disadvantage of a weak robustness.

[103] introduced a notion *codiagnosability* to describe a requirement that any failure can be diagnosed within bounded delay by at least one local diagnoser using it's own observations of the system execution. The *codiagnosability* property is stronger than *diagnosability* under the aggregate observations, which declaimed a possibility that a system is centrally diagnosable but not decentrally

diagnosable.

[68, 69] focused on solving the ambiguity of several local diagnosis towards the global diagnosis (introduced and discussed in [134] and its extended version [135]). [69] proposed a framework for performing diagnosis in a decentralized setting. A global diagnosis decision is taken to be a *winning* local diagnosis decision, which is tagged with a certain ambiguity level. The work showed that the codiagnosability introduced in [103] was the same as 0-inference-diagnosability; the conditional codiagnosability introduced in [134] and [135] was a type of 1-inference-diagnosability; and the class of higher-index inference-diagnosable systems strictly subsumed the class of lower-index ones. The author of [135] claimed their architecture can be realized in a distributed environment.

2.5.3 Distributed diagnosis

In a distributed diagnosis environment, the overall system consists of different components, and associated with each component, there is a local agent (diagnoser-agent) that derives the local diagnosis of its component. Each local agent only knows the model of the local component and of its interactions with its neighbors. Each local agent moreover only receives signals from the monitoring system for the local events. No centralized structure is assumed to coordinate the results of the local agents but from time to time the local agents may exchange messages over communication channels linking them. Thus the local agents derive the distributed diagnosis by local calculations and by information exchanges, e.g., [124, 56, 125, 59, 37, 118, 57, 105, 36].

Generally speaking, distributed architectures for diagnosis differ from decentralized ones in terms of the local models used at the different sites for model-based inferencing and in terms of the ability for local diagnosers to communicate among each other in real-time.

For a distributed system without coordinator, the consistency check between the local diagnoses is merely important. The *local consistency* requires that all local diagnoses agree on their mutual interfaces. While the *global consistency* ([123]) requires the local diagnoses are the projected versions of the global diagnosis, which needs a global system model. Some works, e.g., [101] transform the topology of the system into a junction tree where each vertex represents a subsystem. Local consistency between the diagnoses of these subsystems ensures global consistency due to the tree structure.

[104], extended from [103] defined a new observation mask for each local observer that combined the effect of it's own observation and the bounded-delay communication received by other diagnosers. The local diagnosers communicate with each other using the *immediate observation passing protocol*. Thus the distributed diagnosis problem is reduced to a decentralized diagnosis

one. The further extended work [105] reduce the complexity of on-line diagnosis at each local site to be linear with the number of sites by proposing a new distributed diagnosis protocol.

[124, 125, 123] proposed an automaton-based distributed and hierarchal diagnosis architecture. Each local component has its own local diagnoser, which is built based only on knowledge about this component. The stored size of the overall diagnoser is only the sum of state sizes of the local diagnosers, hence spatial complexity is kept under control. Each local diagnoser is connected with other local diagnosers based on the input/output relations among associated local components. Adding new components, taking components out of the system or changing the input/output relations among local components only affects the local diagnosers that are directly associated with the altered components. A hierarchical computational procedure and multi-resolution diagnosis approach are introduced in [123] to overcome the shortcomings of high time complexity and poor scalability of the distributed ones.

[110, 35, 37, 36, 39] discussed the distributed diagnosis problem based on PN model.

[37, 36] based on the work of [110, 12] discussed the distributed monitoring and diagnosis problems of the asynchronous subsystems with partial ordered observations. The idea here is that when concurrent subsystems are composed, there may be events in the alphabets of the subsystems whose relative order is not important. Therefore, partial-order techniques reduce the complexity of a model by not capturing all the permutations of the orderings of these events. [36] proposed and discussed different kinds of data structures (execution tree, unfolding, trellis, etc.) of representing the asynchronous communication, real concurrency and partial ordered events for the distributed diagnosis. The necessariness of defining partial order is strongly insisted in this work.

[35] modeled a distributed system as a graph of interacting subsystems, with the appropriate semantics of trajectories and stochastic framework. A centralized supervisor, collecting all observation from the system and knowing a model of the whole system, may not be affordable, so they advocate instead a processing by parts, and extend the idea towards a completely distributed supervisor architecture, with one local supervisor on the top of each subsystem, coordinating its activity with the supervisors in its neighborhood.

In [118], distributed diagnosis for Petri nets with synchronous communication is studied. The authors extend the notion of DES¡¯s diagnosers to PN¡¯s and centralized and distributed diagnosers are designed. The centralized approach presents the same problems of combinational explosion than the original based on FSM and the distributed approach focuses on the problem of communication between the diagnosers.

[58] proposed a distributed diagnosis based on place-boarded PN models, which are bounded,

ordinary and known initial marking. A fault in a PN model is represented by a choice transition. The case of unobservable interactions between components and cyclic communications are considered. The minimal explanations are derived by backward inference on each local diagnoser based on the local partial ordered observations. [58] concerned the diagnosis of plant systems, so the system model is assumed to be global clock scheduled instead of event-driven. A local diagnoser first searches for the *minimal configuration* to decide the initial marking with *backward unfolding* approach then infers forwardly for the possible local exited tokens to update the state of its neighbor diagnosers. The global consistency is verified by comparing the causal relations of the communicating events between the different sites with the observations. The state explosion problem is partially controlled with partial observation.

2.6 Conclusion

The correctness and efficiency of MBD depends mainly on three elements: the system model, the fault representation, and the diagnosis approach. On the level of system model, LTS (automata) is suitable for monotonic system with smaller states set and larger events set; PN is suitable for real concurrent system. On the level of fault representation, faulty states and events are normally adopted separately. On the level of diagnosis approaches, the diagnoser, unfolding, and (backward) reachability approaches all suffer a lot from synchronization. The PN algebraic approach can help to improve the efficiency while the fault representation becomes difficult and complicated.

While there is no absolute adequate standard, the final choice depends on the system characteristics and the aim of diagnosis.

The thesis dedicates to diagnose the dysfunctions of large software systems, as discussed in chapter 1, which has the following properties:

1. Besides unobservable faulty events, faulty input can be the source of faults *alone*;

2. Except in a real concurrent model, the complicate data flows, such as the parallel and cyclic ones, cause the state space explosion;

3. The calculation of global model and observation should be avoid because of the authorizing limits and asynchronous clocks of the subsystems;

4. Effective on-line diagnosis is preferred but the complexity off-line diagnoser generation should be under control because of item 2;

5. A decentralized diagnosis architecture is preferred because of item 3.

Based on the above analysis, a model-based diagnosis approach with following properties can be an adequate choice:

1. both correct/faulty data and data propagation events are directly represented in the model;

2. the control of the data flow could be modeled in similar way;

3. the model should support different topological architectures.

So in the chapter 3, we introduce a Colored Petri net model which represents both the correct and faulty data (places) and events (transitions). The data dependency relation is introduced as an arc expression to represent the data propagation. The control of the data flow is represented with activation places, and the algebraic diagnosis approach is used to calculate the minimal diagnosis based on the data propagation and incidence matrix equation.

Part I

Theory: a CPN model for diagnosis in a distributed environment

Chapter 3

Colored Petri net model for MBD

3.1 Introduction

Colored Petri net (CPN) [70] is an extension of PN with the data type (named as *color set*) definition on the places of PN. The creation and development of CPN was driven by the requirement of an abstract model which is theoretically well-founded and versatile enough to represent the large and complex industrial systems. The CPN model combines the strength of Petri nets, the synchronization of concurrent processes, with that of the programming languages, data types definition and data values manipulation.

Comparing to other formal models, like process algebra, LTS, automata, or PN, CPN model has the following advantages for diagnosing the composite systems:

- CPN is more compact than PN thanks to the ”*color*” definition;
- CPN model integrates both data manipulations and process control (including synchronization, concurrency, and structure hierarchy);
- CPN supports transition mode definition which makes the fault representation more flexible;
- CPN keeps the mathematical properties, formal analysis methods of PN like incidence equation, state space place invariant analysis, and the analysis is supported by the powerful CPN tools [54, 84, 137].

With all the above complicate properties, CPN is in particular well suited for the systems emphasize in communication, synchronization, and resource sharing. The classical applications are

communication protocols [60, 28, 86], distributed systems [81, 22], imbedded systems [89], power systems [27, 112], automated production/conytol systems [127, 62], work flow analysis [16], and VLSI chips [119]. The readers are referred to [67] for more details.

3.2 Structure and dynamic

3.2.1 Structure of CPN

In comparison with classical Petri nets the Colored Petri Net (CPNs) introduces the notion of token types, namely tokens are differentiated by colors, which may be arbitrary data values. Each place has an associated type determining the kind of data that the place may contain. The marking of a given place is a multi-set of values of the associated type. Arcs constraint are expressions that extract or produces multi-sets by respect to the sources or target types. For CPNs we use terms like types, values, operation, expression, variable, binding and evaluation and they have the same meanings as in programming languages. In order to give a definition of the CPN, we give here, without loose of generality, a simple syntax and semantic for expression.

- Types : noted by Π, we range over by using π_i. Types are defined by the set of values that compose it, $\pi = \{\nu_0, ..., \nu_i, ...\}$. Also types can be defined by applying set operation on types.
- Variables : noted by $\mathcal{X}$, we range over by χ_i. Variables are typed and as usual we use $Type(\chi)$ to obtain the type of χ.
- Function : denoted by $\mathcal{F}$, for a function $f \in \mathcal{F}$ with $f : \pi \to \pi'$ we use $Type(f)$ to define its range type.

Definition 31 (Multi-set)**.** *Let E be a set, a* multi-set *m on E is an application $m : E \to \mathbb{Z}$ (we use the formal sum notation for a multi-set, $m = \sum_{0<i\leqslant n} q_i^{\backprime} e_i, n > 0$ where $q_i = m(e_i)$). We use $\mathcal{M}(E)$ to define the set of finite multi-sets from E to $\mathbb{Z}$, and $\mathcal{M}^+(E)$ if we restrict it to $\mathbb{N}$. Sum and subtract operators between two multi-sets are defined as in [64].*

Note that for a type π, $\mathcal{M}(\pi)$ is the type of multi-set of π values.

Definition 32 (Multi-set expression)**.** *Let π be a types, f a function and χ a variable, a multi-set expression ψ is defined as follows:*

- $\psi ::= \nu \in \pi | \chi | f(\nu, \cdots, \chi, \cdots) | \sum_{0<i\leqslant n} q_i^{\backprime} \psi_i$

We use Ψ to denote the set of expression and we range over using ψ_i, and $\Psi(\pi)$ is the expression set of type π.

For a given multi-set expression $\psi \in \Psi$ we define the following notations :

- $Var(\psi)$ denote the set of variables that appears in ψ.
- A binding β of a set of variables V is an association to each variable $\chi \in V$ a value of $Type(\chi)$ ($\beta(\chi) \in Type(\chi)$).
- ψ^β denote the valuation of ψ under the binding β.
- $Type(\psi)$ denote the type of the expression with $Type(\psi) = \pi$ iff $\forall\beta, \psi^\beta \in \mathcal{M}(\pi)$. $\Psi\langle\pi\rangle$ is used to denote the set of expression of type π

Here after we give a general definition of CPN in n-uplet format. In our definition we consider both places and transition types. We call colors the types of places and modes the types of transitions. Also we are in the case of *open expression* arcs annotation (in opposition to *constant arcs expression*).

Definition 33 (CPN graph)**.** *A Colored Petri Net graph (CPN graph) is a tuple N=$\langle\Sigma$, Γ, P, T, cd, Pre, $Post\rangle$, where:*

(i) Σ *is a set of no-empty types, also called* color sets*;*

(ii) Γ *is set of mode types;*

(iii) P *is a set of labeled places;*

(iv) T *is a set of labeled transitions;*

(v) $cd : P \to \Sigma$*;*

(vi) $md : T \to \Gamma$ *with* $md(t) = t \cdot m$*;*

(vii) Pre *(resp. post)* $\in \mathbb{B}^{|P|\times|T|}$ *:, where* $\mathbb{B} = \bigcup_{\sigma\in\Sigma} \bigcup_{\gamma\in\Gamma} [\gamma \to \Psi\langle\sigma\rangle]$[1]*, are forward (resp. backward) matrices with* $Pre[p, t]$ *(resp.* $Post[p, t]$*):* $md(t) \to \Psi\langle cd(p)\rangle$*.*

[1] we use the notation $[D \to D']$ to represent the set of all application from D to D'.

(i) The set of types determines the data values and the operations and functions that can be used in the net expressions (i.e., arc expressions). If desired, the types (and the corresponding operations and functions) can be defined by means of a many-sorted *sigma algebra* (as in the theory of abstract data types). We assume that each type has at least one element (see example 5).

(v) The color function cd maps each place, p, to a type $cd(p)$. Intuitively, this means that each token on p must have a data value that belongs to $cd(p)$.

(vii) Each expression of type $\mathbb{B}$ must evaluate to multi-sets over the type of the adjacent place, p. We allow a CPN diagram to have an expression $expr$ of type $cd(p)$, and consider this to be a shorthand for $1`expr$. [64]. While we consider transition mode each input (resp. output) arc between a place p and transition t ($Pre[p,t] \neq \emptyset$ resp. $Post[p,t] \neq \emptyset$), we will have as many extraction (resp. production) expressions as modes in the transition t. That explains the format of Pre (resp. $Post$) matrix which is defined on functions from the product of places and transitions modes type to the multi-set expressions. $Pre[p,t]$ (resp. $Post[p,t]$) is a vector of $\Psi\langle cd(p)\rangle$ indexed by the modes of t (denote as $t \cdot m$).

3-d-p example 5. *A CPN graph with 2 places p_1, p_2, each of which has two different colors, and 1 transition t_1, in which t_1 has two modes γ_1 and γ_2.*

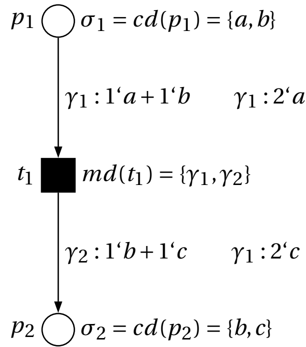


Figure 3.1: A CPN graph example

Pre	t_1	
	γ_1	γ_2
p_1	$1`a + 1`b$	$1`b + 1`c$
p_2		

$Post$	t_1	
	γ_1	γ_2
p_1		
p_2	$2`a$	$2`c$

Table 3.1: Pre and $Post$ matrixes

We extend here the usual matrices notations:

- $Pre[.,t]$ (resp. $Post[.,t]$) is sub matrix of Pre (resp. $Post$) Matrix obtained by the projection of the columns only on t modes.

- $Pre[p,.]$ (resp. $Post[p,.]$) is a row vector of $cd(p)$ expressions indexed by the union of all the transition modes.

For a given transition t and one of its mode $t \cdot m$, we use the following notation :

- $Pre[p, t \cdot m]$ (resp. $Post[p, t \cdot m]$) for $Pre[p, t](t \cdot m)$ (resp. for $Post[p, t](t \cdot m)$)
- $Pre[., t \cdot m]$ ((resp. $Post[., t \cdot m]$) for $Pre[., t][., t \cdot m]$ (resp. for $Post[., t][., t \cdot m]$)

While $Pre[., t \cdot m]$ is vector of expression we extends $Var(Pre[., t \cdot m]) = \bigcup_{p \in P} Var(Pre[p, t \cdot m])$. Also for a binding β $Pre[., t \cdot m]^\beta = (..., \psi_i^\beta, ...)$ for $i = 1...|P|$ is the resulted vector after the application of β for each expression of $Pre[., t \cdot m]$

Definition 34 (Well-formed CPN Graph)**.** *A CPN graph* $N = \langle \Sigma, \Gamma, P, T, cd, md, Pre, Post \rangle$ *is* well formed *iff:*

- $\forall t \in T, p \in t^\bullet, t \cdot m \in md(t) : Var(Post[p, t \cdot m]) \subseteq Var(Pre[., t \cdot m])$ *with* $Var(Pre[., t \cdot m]) = \bigcup_{p' \in {}^\bullet t} Var(Pre[p', t \cdot m])$.
- $\forall p, t : \exists t \cdot m \in md(t), Pre[p, t \cdot m] \neq 0 \Rightarrow \forall t \cdot m' \in md(t), Pre[p, t \cdot m'] \neq 0$.
- $\forall p, t : \exists t \cdot m \in md(t), Post[p, t \cdot m] \neq 0 \Rightarrow \forall t \cdot m' \in md(t), Post[p, t \cdot m'] \neq 0$.

For a given transition and one of its mode we restrict that the output arc expressions variables must be a subset of the variables which are in the input arcs expressions of the same mode. modes represents the scopes of variable names. In addition we impose that modes respect the structure of the Petri Net ; if a palace is an input place of transition so it is for all its modes. Thus, we use the usual notation ${}^\bullet t$ and $t^\bullet$ for the input and output places set of transition t and ${}^\bullet p$ and $p^\bullet$ for the input and output transitions set of place p.

Definition 35 (Incidence Matrix of CPN)**.** *To each CPN graph, we associate its terms incidence Matrix* $C = Post - Pre$ *(see table 3.2).*

Definition 36 (Fault model of CPN)**.** *The* fault model *of a CPN is a function* FM*:* $\bigcup_{\gamma_i \in \Gamma} \rightarrow \{OK, KO\}$ *(i.e., item 2 of example 6).*

3-d-p example 6. *The dining philosophers in form of CPN (in figure 3.2) can be defined as follows:*

1. $\Sigma = \{F = \{①, ②, ③\}\}$*: The type forks (*$F$*) contains three series numbers of forks.*
2. $\Gamma = \{WO = \{co\}, UO = \{co, sw\}\}$*: two types of transitions; the Well-Organized (WO) type which has only one mode co (Correctly put the forks down), correspond to the classical transition, and the Un-Organized (UO) type which behaves either correctly (co) or switches the forks (*sw*) .*

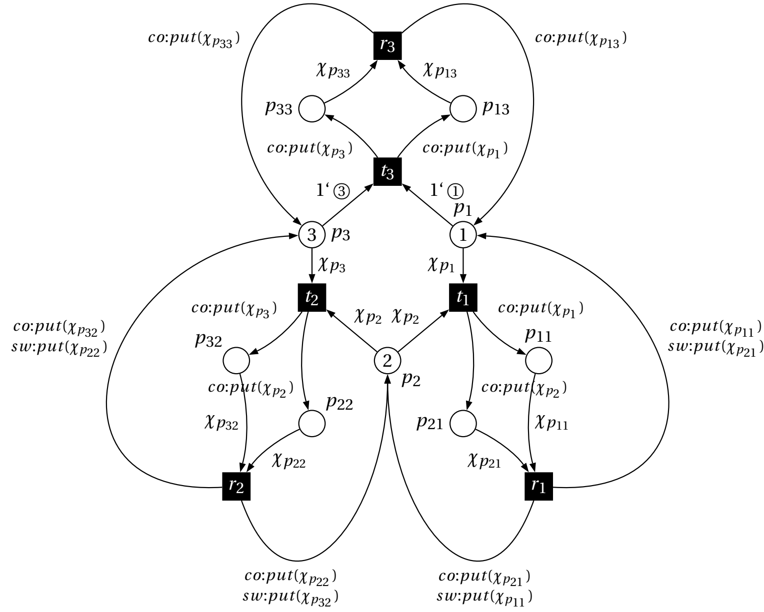


Figure 3.2: Dining philosophers: CPN model

3. $P = \{p_1, p_2, p_3, p_{11}, p_{12}, p_{22}, p_{23}, p_{33}, p_{31}\}$*:* p_i *is the series number of plate on the right side of philosopher* i *before he/she takes the forks. Places* p_{ii} *and* p_{ji} *(*$j \neq i$*) represent the intermediate places after philosopher* i *takes the forks.*

4. $T = \{t_1, t_2, t_3, r_1, r_2, r_3\}$*:* t_i *and* r_i*is for the take (both forks) transition of the philosopher* i *and* r_i *is for the release (of both) transition of philosopher* i*.*

5. *Variables* χ_i *on the input arc of transitions* t_i *represent the fork series numbers.*

6. *Variables* χ_{ii} *and* χ_{ji} *(*$j \neq i$*) on the output arc of transitions* t_i *represent the fork series numbers in the right and left hands of philosopher* i *after he/she takes the forks.*

7. $\forall p \in P, cd(p) = F$*.*

8. *if* $t \in \{r_1, r_2\}$ *then* $md(t) \in UO$ *else* $md(t) \in WO$*.*

9. *A function put:* $\Sigma \rightarrow \Sigma$ *represents the action "putting down the forks" after a philosopher eats. For* $md(r_1) \in UO$ *and* $md(r_2) \in UO$*, the arc expressions on the left and right output arcs of* r_1 *and* r_2 *are reversed from the input ones.*

Table 3.2 illustrates the incidence matrix of the 3-d-p example. Each table element represents

an arc expression which is either an expression, variable, or function of multi-set. The blank ones represent there is no arc.

C	t_1	t_2	t_3	r_1		r_2		r_3
	$t_1 : co$	$t_2 : co$	$t_3 : co$	$r_1 : sw$	$r_1 : sw$	$r_2 : co$	$r_2 : sw$	$r_3 : co$
p_1	$-\chi_{p_1}$		$-\chi_{p_1}$	$put(\chi_{p_{11}})$	$put(\chi_{p_{21}})$			$put(\chi_{p_{31}})$
p_2	$-\chi_{p_2}$	$-\chi_{p_2}$		$put(\chi_{p_{21}})$	$put(\chi_{p_{11}})$	$put(\chi_{p_{22}})$	$put(\chi_{p_{32}})$	
p_3		$-\chi_{p_3}$	$-\chi_{p_3}$			$put(\chi_{p_{32}})$	$put(\chi_{p_{22}})$	$put(\chi_{p_{33}})$
p_{11}	$put(\chi_{p_1})$			$-\chi_{p_{11}}$	$-\chi_{p_{11}}$			
p_{21}	$put(\chi_{p_2})$			$-\chi_{p_{21}}$	$-\chi_{p_{21}}$			
p_{22}		$put(\chi_{p_2})$				$-\chi_{p_{22}}$	$-\chi_{p_{22}}$	
p_{32}		χ_{p_3}				$-\chi_{p_{32}}$	$-\chi_{p_{32}}$	
p_{31}			χ_{p_1}					$-\chi_{p_{31}}$
p_{33}			χ_{p_3}					$-\chi_{p_{33}}$

Table 3.2: Dining philosophers: incidence matrix

3.2.2 Dynamic of CPN

Dynamic properties characterize the behavior of individual CPN, e.g., whether it is possible to reach a marking in which no step is enabled. In the following, we define the behaviors (the dynamics) of a CPN System.

Definition 37 (CPN marking)**.** *A* marking M *of a CPN graph is a multi-set vector indexed by* P*, where* $\forall p \in P$*,* $M(p) \in \mathcal{M}^+(cd(p))$*. We use* $\mathbb{M}_N$ *to denote the set of all marking of a net* N*.*

Definition 38 (CPN system)**.** *A* Colored Petri Net System *(CPN-S) is a pair* S=$\langle N, M\rangle$ *where* N *is a CPN graph and* M *is one of its marking.*

Definition 39 (CPN mode enabling rules)**.** *Let* S=$\langle N, M\rangle$ *be a CPN system,* t *be a transition in* N *and* m *be one of its modes,* $m \in md(t)$ *:*

- *A mode* $t \cdot m$ *is enabled, noted* $M[t \cdot m\rangle$*, iff* $\exists\beta$*, with* $M \geq Pre[., t \cdot m]^\beta$*.*
- *A transition* t *is enabled, noted by* $M[t\rangle$*, iff* $\exists t \cdot m \in md(t), M[t \cdot m\rangle$*.*

Definition 40 (CPN mode firing rules)**.** *Let* S=$\langle N, M\rangle$ *be a CPN-S,* t *a transition and* m *one of its mode, with* $M[t \cdot m\rangle$ *for some* β*. The firing of the transition* t *under mode* m *changes* S *to* $S' = \langle N, M'\rangle$ *with* $M' = M + C(., t)(t \cdot m)^\beta$*. We denote the firing as* $M[t \cdot m\rangle^\beta M'$ *and some times we can abstract from the binding.*

Definition 41 (CPN mode sequence firing rules)**.** *We extend the definition 40 to a sequence of modes $\delta \in Mod^*$ (with $Mod = \bigcup_{t \in T} md(t)$) in usual way :*

- *$M[\delta\rangle M$ if δ is the empty sequence;*
- *$M[\delta\rangle M'$ if $\exists M'' \in \mathbb{M}, \delta' \in Mod^*, t \in T, t{\cdot}m \in md(t)$ such that $M[\delta'\rangle M''$ and $M''[t{\cdot}m\rangle M'$ with $\delta = \delta' \cdot m$*

Definition 42 (CPN reachability)**.** *Let $S{=}\langle N, M\rangle$ be a CPN system, we note the set of reachable marking from M, $[M\rangle = \{M' \in \mathbb{M} | \exists \delta \in Mod^*, M[\delta\rangle M'\}$*

Definition 43 (CPN characteristic vector)**.** *Let $\delta \in Mod^*$ be a sequence of modes of a net, its* characteristic vector *$\overrightarrow{\delta} : Mod \to \mathbb{N}$* [2] *with $\overrightarrow{\delta}(m)$ is the number of occurrence of m in δ. For each δ, we associate a transition sequence $\delta_T \in T^*$ s.t. $\forall i \in 1, \cdots, |\delta|$, $T[i] = md^{-1}(\delta_T[i])$. So its characteristic vector is $\overrightarrow{\delta_T} : T \to \mathbb{N}$. Note that: $\overrightarrow{\delta_T}(t) = \sum_{t \cdot m \in md(t)} \overrightarrow{\delta}(t \cdot m)$.*

3-d-p example 7. *A characteristic vector $\overrightarrow{\delta}$ (see table 3.3) which means philosopher 1 ate twice, once put the forks down correctly but another time wrong ($\overrightarrow{\delta}(t_{1_{co}}) = 2$, $\overrightarrow{\delta}(r_{1_{co}}) = 1$, and $\overrightarrow{\delta}(r_{1_{sw}}) = 1$); philosopher 2 ate normally ($\overrightarrow{\delta}(t_{2_{co}}) = 1$, $\overrightarrow{\delta}(r_{2_{co}}) = 1$, and $\overrightarrow{\delta}(r_{2_{sw}}) = 0$), thus philosopher 3 stopped eating when he/she found the fault ($\overrightarrow{\delta}(t_{3_{co}}) = 1$ and $\overrightarrow{\delta}(r_{3_{co}}) = 0$). The corresponding characteristic vector of transition sequence $\overrightarrow{\delta_T}$ (see table 3.4) is a vector of the occurrence sums of all the modes of each transition.*

$t_{1_{co}}$	$r_{1_{co}}$	$r_{1_{sw}}$	$t_{2_{co}}$	$r_{2_{co}}$	$r_{2_{sw}}$	$t_{3_{co}}$	$r_{3_{co}}$
⟨ 2	1	1	1	1	0	1	0 ⟩

Table 3.3: Dining philosophers: a characteristic vector $\overrightarrow{\delta}$.

t_1	r_1	t_2	r_2	t_3	r_3
⟨2	2	1	1	1	0⟩

Table 3.4: Dining philosophers: a characteristic vector of a transition sequence $\overrightarrow{\delta_T}$.

Given $S = \langle N, M\rangle$ a CPN-S and modes sequence, $\sigma \in Mod^*$ with $M[\sigma\rangle$, then the reached marking M' after the firing of σ is $M' = M + C \times \overrightarrow{\sigma}$.

[2] $\rightharpoonup$ represents a vector, which is a column of elements of the same type domain

3-d-p example 8. *Given an initial marking M_0 and a characteristic vector $\overrightarrow{\delta_T}$ (see table 3.3), know the incidence matrix C (see table 3.2), we can calculate the final marking M_n of the CPN-s based on the equation above (see table 3.5).*

M_n	=	M_0	+	$C\times$	$\overrightarrow{\delta_T}$
p_1: $\emptyset$		1`①			$t_1 \cdot co$: 2
p_2: 1`①		1`②			$t_2 \cdot co$: 1
p_3: $\emptyset$		1`③			$t_3 \cdot co$: 1
p_{11}: $\emptyset$		$\emptyset$			$r_1 \cdot co$: 1
p_{21}: $\emptyset$	=	$\emptyset$	+	$C\times$	$r_1 \cdot sw$:1
p_{22}::$\emptyset$		$\emptyset$			$r_2 \cdot co$: 1
p_{32}: $\emptyset$		$\emptyset$			$r_2 \cdot sw$:0
p_{31}: 1`②		$\emptyset$			$r_3 \cdot co$: 0
p_{33}: 1`③		$\emptyset$			

Table 3.5: Dining philosophers: incidence equation

3.3 CPN as a fault model for software systems

As claimed before, the thesis focuses on the diagnosis of the software systems which means the systems are composed by data and the activities over data. CPN is a rich model usually used to design and check such software systems. Mainly, when using CPN for software modeling, data are represented as places while transitions are used for activities. The CPN structure also codes the data dependency by the arcs and also the nature of the dependency by the arcs expressions. In this thesis, the CPN model is used to define a fault model for the software systems. This basic idea of using CPN as fault model are as follows:

- using places to store data, transition to represent the activities, and arcs expressions to represent the data dependency under different transition modes;
- using the places types to represent the data status like corrupted data, correct data, etc.
- using transition modes to represent the corrupted and correct activities;
- profiting from the CPN properties to perform diagnosis;
- finally, defining a method that generates semi-automatically a CPN fault model directly from the CPN model of a system.

In the sequel of this section, we details in the different parts that define the CPN fault model and we finish this section by giving a method to transform any system to a CPN model as a fault model.

3.3.1 The CPN fault model structure

The CPN as fault model has the same structure as any CPN except with some restriction on the places types, the arcs expressions, and the transition modes. Here after we motivate such restrictions.

Places types: data status

When a system crashes due to some wrong variable value, the diagnosis consists in locating the *variable* that causes the crash but not its real *value*. Consider the 3-d-P example, the philosopher number 3 expects the fork number 1 in his left when he founds the fork number 3 or number 2, he believes the place 1 has a corrupted fork value. So for each place no matter what the type of data is, when performing diagnosis, we focus only on its correctness status. Thus in the CPN fault model, all the places share the same type: the color status type. The places color status type is defined by three values to represent the correctness status of each token (data):

- red (r) marks a place with faulty data value;
- black (b) marks a place with correct data value;
- unknown color ($*$) marks a place with the data value of unknown correctness.

We note the color status type as $\Sigma = \{status = \{r, b, *\}\}$.

Of course to respect the quantitative constraints that define the dynamic of a CPN some places can be typed by a multi-set of Σ

Arcs expressions: abstract data dependency

To specify the causality between the input and output places in CPN, we use the multi-set expression on the input and output arcs of each transition. Again, when dealing with diagnosis, the value of the retrieved and produced tokens is not relevant. If a variable contains corrupted data, diagnosis is to decide if it is possible that the corruption is due to the update of a variable by using other corrupted data. So we need in a CPN fault model to code two information, the dependencies relations between data and the nature of that dependencies. In our CPN model, the dependency between data is naturally represented by the precedent structural relations (Pre and $Post$). The nature of dependency is defined by the multi-set expression over arcs.

For the input arcs we restrict that the multi-set expressions are only over status color constants and color variables, no function is allowed. This allows to define the quantitative constraints that defines the dynamic of the CPN and the abstraction from the real data values (only the number and the status are important). Let $\Psi_{\mathcal{C}}^{Pre}$ be the set of multi-set expressions defined only on the status color constants and variables, the pre-condition matrix of CPN fault model is $Pre \in \mathbb{B}^{|P|\times|T|}$: where $\mathbb{B} = \bigcup_{\gamma\in\Gamma}[\gamma \to \Psi_{\mathcal{C}}^{Pre}]$ (i.e., see the arc expressions of $p_1 \to t_3$ and $p_{13} \to r_3$ in figure 3.2 which are respectively 1' ①and $\chi_{p_{13}}$).

Concerning the output arcs, we are interested to store the nature of dependency between the produced output tokens and the consumed input tokens. Three abstract dependency functions are defined as follows:

Definition 44 (FW function)**.** *A* FW *function* $FW : X \to X$ *with* $\forall x \in X$, $FW(x) = x$.

Definition 45 (SRC function)**.** *A* SRC *function* $SRC : \emptyset \to X$.

Definition 46 (EL function)**.** *An* EL *function* $EL : X^n \to Y$ *with* $n \in \mathbb{N}$.

So FW and SRC functions are two special cases of EL function.

- When an output token coincides with an input parameter, we say that the transition *forwards* (FW for short) the input to the output;
- When an output parameter is created during an activity, we say the activity is the *source* (SRC) for it.
- When an output parameter is computed by the activity from one or more inputs, we say that it is the result of an *elaboration* (EL).

Let $\Psi_{\mathcal{C}}^{Post}$ denotes the multi-set defined over the status constants, status variables, and the three classes of functions FW, SRC, EL. The post-condition matrix of CPN fault model is $Post \in \mathbb{B}^{|P|\times|T|}$: where $\mathbb{B} = \bigcup_{\gamma\in\Gamma}[\gamma \to \Psi_{\mathcal{C}}^{Post}]$.

Transition modes: faults

Generally in MBD using DES, faults are represented as unobservable events which means something goes wrong that we cannot observe. When dealing with the software systems, such representation of fault can be far from the reality. To explain our purpose, we suppose a function f to divide

a natural number by 2. When we input an odd number, we get a correct answer and otherwise we go a wrong result. In fact, f implements an entire division. Here we define the dysfunction as an unobservable event that occur each time with an even number. Also the functions can be interpreted as working in two modes: a correct mode $f : OK$, or a faulty mode $f : KO$, but only f is observed. So we don't violate the parsimony principal that the possibly faulty transition can act under two modes, which we cannot observe. Given an observation of transitions, the diagnosis consists in determining the modes of transitions that explain the symptoms.

In CPN fault model, we keep the modes types which conform to the correct and faulty behaviors of the system. We add also a flag function F that maps to each mode its correct (OK) or fault (KO) status. So for a set of modes types Γ, we define $F : \bigcup_{\gamma \in \Gamma} \gamma \rightarrow \{OK, KO\}$.

CPN fault model definition

We can now give the definition of CPN Fault Model graph.

Definition 47 (CPN fault model graph)**.** *A CPN is a tuple* N=$\langle\Sigma$, Γ, P, T, Pre, $Post$, F,$\rangle$, *where:*

(i) Σ *is the status colors type;*

(ii) Γ *is set of mode types;*

(iii) P *is a set of labeled places of type* Σ*:* $\forall p \in P$, $cd(p) \in \Sigma$*;*

(iv) T *is a set of labeled transitions;*

(v) $md : T \rightarrow \Gamma$*;*

(vi) $Pre \in \mathbb{B}^{|P|\times|T|}$ *: is the forward matrices, where* $\mathbb{B} = \bigcup_{\gamma \in \Gamma} [\gamma \rightarrow \Psi\langle\sigma\rangle]$*;*

(vii) $Post \in \mathbb{B}^{|P|\times|T|}$ *: is the backward matrices;*

(viii) $F : \bigcup_{\gamma \in \Gamma} \gamma \rightarrow \{OK, KO\}$.

3-d-p example 9. *Figure 3.3 illustrates the example with the abstract data dependency relations. Note that for the transitions under* OK *mode, the abstract data dependency is* FW *as output is equal to input, while for those in* KO *modes, the data dependency is* EL*.*

Table 3.6 illustrates the incidence matrix of the example with the abstract data dependency. Each table element represents an arc expression which is either an expression, variable, or abstract data dependency of multi-set.

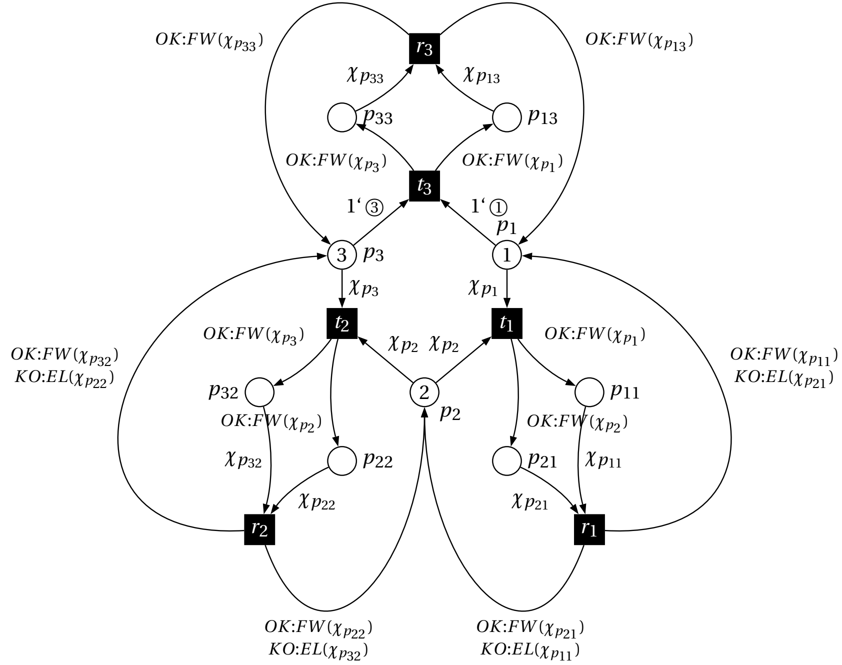


Figure 3.3: Dining philosophers: CPN with abstract data dependency relations

C	t_1	t_2	t_3	r_1		r_2		r_3
	$t_1 \cdot OK$	$t_2 \cdot OK$	$t_3 \cdot OK$	$r_1 \cdot OK$	$r_1 \cdot KO$	$r_2 \cdot OK$	$r_2 \cdot KO$	$r_3 \cdot OK$
p_1	$-\chi_{p_1}$		$-\chi_{p_1}$	$FW(\chi_{p_{11}})$	$EL(\chi_{p_{21}})$			$FW(\chi_{p_{31}})$
p_2	$-\chi_{p_2}$	$-\chi_{p_2}$		$FW(\chi_{p_{21}})$	$EL(\chi_{p_{11}})$	$FW(\chi_{p_{22}})$	$EL(\chi_{p_{32}})$	
p_3		$-\chi_{p_3}$	$-\chi_{p_3}$			$FW(\chi_{p_{32}})$	$EL(\chi_{p_{22}})$	$FW(\chi_{p_{33}})$
p_{11}	$FW(\chi_{p_1})$			$-\chi_{p_{11}}$	$-\chi_{p_{11}}$			
p_{21}	$FW(\chi_{p_2})$			$-\chi_{p_{21}}$	$-\chi_{p_{21}}$			
p_{22}		$FW(\chi_{p_2})$				$-\chi_{p_{22}}$	$-\chi_{p_{22}}$	
p_{32}		χ_{p_3}				$-\chi_{p_{32}}$	$-\chi_{p_{32}}$	
p_{31}			χ_{p_1}					$-\chi_{p_{31}}$
p_{33}			χ_{p_3}					$-\chi_{p_{33}}$

Table 3.6: Dining philosophers: incidence matrix with abstract data dependency

3.3.2 The CPN fault model dynamic

The concepts defined for the CPN are steel valid for CPN fault model such as bindings, firing rules, production rules etc. Except that we did'nt give yet a semantic to the abstract dependency functions under a correct or faulty mode. The semantic of the FW, EL, and SRC is given based on the color propagation execution.

Tables represent the diagnosis properties of each abstract data dependency relation of I/O of a transition for different modes (OK, KO). In each table, m_t represents the transition mode, $c_{\bullet t}/c_{t\bullet}$ represents the color of the token in the input/output place.

As illustrated in table 3.7c, the diagnosis properties of FW do not change along with the modes: the correctness of output is always same with that of the input. For EL (see table 3.7a) and SRC (which can be seen as ELs with no (unknown) input, see table 3.7b), the diagnosis properties can be stated as:

- in the *OK* mode, if one of the inputs is faulty (r), the output must be faulty (r), if all the inputs are either correct (b) or unknown ($*$), the output is unknown ($*$), while if the inputs are all correct (b) then also the output is correct;
- in the *KO* mode, the output is faulty (r) if there is no unknown ($*$) input, otherwise, the output is unknown ($*$).

So the output of SRC relation is correct (b) under OK mode and faulty under KO mode ($*$).

EL			
m_t	$c_{\bullet_i t}$	$c_{\bullet_j t}$	$c_{t\bullet}$
OK	b	b	b
OK	r	b	r
OK	$*$	b	$*$
OK	$*$	$*$	$*$
OK	$*$	r	r
KO	b	b	r
KO	r	b	$*$
KO	$*$	b	$*$
KO	$*$	$*$	$*$
KO	$*$	r	$*$

(a) EL

FW		
m_t	$c_{\bullet t}$	$c_{t\bullet}$
OK/KO	b	b
OK/KO	r	r
OK/KO	$*$	$*$

(b) FW

SRC	
m_t	$c_{t\bullet}$
OK	b
KO	r

(c) SRC

Table 3.7: I/O data dependency of m_t for CPN diagnosis: $c_{\bullet_1 t}$ and $c_{\bullet_2 t}$ are two input places of t.

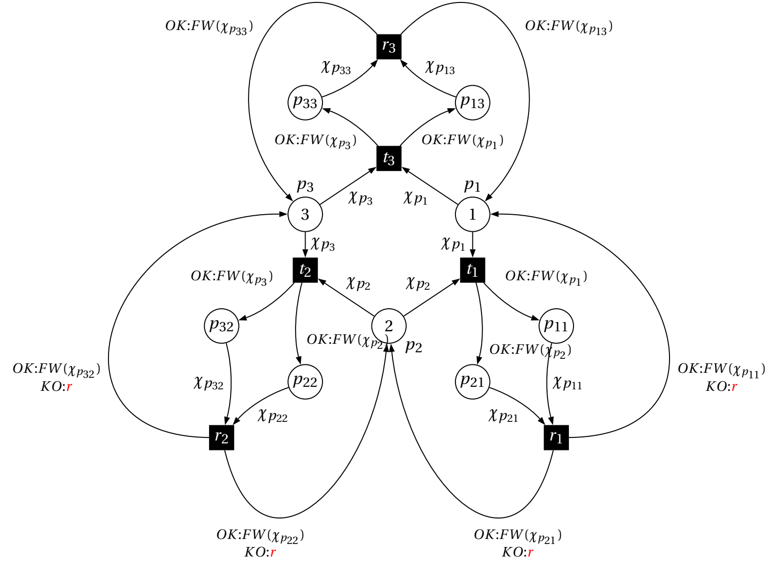


Figure 3.4: Dining philosophers: CPN model for diagnosis

The semantic of each of the abstract data dependency relation is defined based on the color propagation rules to represent the data status (faulty, correct, or unknown status) production. The color propagation function is equivalent to the tables .

With the color set and color propagation function definitions, the diagnosis properties are formally introduced into the literature of CPN.

3-d-p example 10. *Figure 3.4 illustrates the example with the diagnostic data dependency relations. For the transitions in OK mode, the abstract data dependency is FW as output color is equal to input color, while for those in KO modes, the data dependency is EL and the output color is always r.*

Table 3.8 illustrates the incidence matrix of the diagnostic model of the 3-d-p example. Each table element represents a diagnostic arc expression which is either an expression, variable, or color propagation function of multi-set.[3]

3.3.3 Partial observation of CPN fault model

As we explained before, in CPN fault models, correct and fault events are flagged with OK and KO as the modes of each transition. Following the principal of unobservabilitiy of faults, we consider

[3]Variables χ_{p_i}, $\chi_{p_{ii}}$, and $\chi_{p_{ij}}$ ($i \neq j$) represent the diagnostic color set variables here.

C	t_1	t_2	t_3	r_1		r_2		r_3
	$t_1 \cdot OK$	$t_2 \cdot OK$	$t_3 \cdot OK$	$r_1 \cdot OK$	$r_1 \cdot KO$	$r_2 \cdot OK$	$r_2 \cdot KO$	$r_3 \cdot OK$
p_1	$-\chi_{p_1}$		$-\chi_{p_1}$	$FW(\chi_{p_{11}})$	r			$FW(\chi_{p_{31}})$
p_2	$-\chi_{p_2}$	$-\chi_{p_2}$		$FW(\chi_{p_{21}})$	r	$FW(\chi_{p_{22}})$	r	
p_3		$-\chi_{p_3}$	$-\chi_{p_3}$			$FW(\chi_{p_{32}})$	r	$FW(\chi_{p_{33}})$
p_{11}	$FW(\chi_{p_1})$			$-\chi_{p_{11}}$	$-\chi_{p_{11}}$			
p_{21}	$FW(\chi_{p_2})$			$-\chi_{p_{21}}$	$-\chi_{p_{21}}$			
p_{22}		$FW(\chi_{p_2})$				$-\chi_{p_{22}}$	$-\chi_{p_{22}}$	
p_{32}		χ_{p_3}				$-\chi_{p_{32}}$	$-\chi_{p_{32}}$	
p_{31}			χ_{p_1}					$-\chi_{p_{31}}$
p_{33}			χ_{p_3}					$-\chi_{p_{33}}$

Table 3.8: Dining philosophers: the incidence matrix

that only the transitions are observable but not their modes. Also due to the concurrency in the semantic of PN in general, the observed transitions can be partially ordered.

Definition 48 (Partial order observation)**.** *Let N be a CPN (fault model), $\mathcal{T}$ be a multi-set of transitions. A partial order observation of N is a couple $(S(\mathcal{T}), \lhd)$ for some σ_T, where:*

- $S(\mathcal{T}) = \{t_i^k | 1 \leqslant k \leqslant \mathcal{T}(t_i)\}$ *is a set of transitions with k represents the occurrence order of transition t_i;*
- $\lhd \subseteq (S(\mathcal{T}) \times S(\mathcal{T}))$ *is a partial order relation, which is transitive and pre-order relation over $S(\mathcal{T})$ s.t $\forall t \in T$, $\forall i, j \in \{1, \cdots, |\mathcal{T}(t)|\} : (t^i, t^j) \in \lhd^*$.*

3-d-p example 11. *Given a set of transitions sequence $\sigma_T = t_1 r_1 t_2 r_2 t_1 r_1 t_3$, the corresponding $S(\mathcal{T})$ and σ (modes sequence) are listed in table 3.9. The multi-set of the transitions occurrence is $\mathcal{T} = 2`t_1 + 2`r_1 + 1`t_2 + 1`r_2 + 1`t_3$. We have a partial relation $\lhd = \{(t_1^{(1)}, r_1^{(1)})$, $(t_2^{(1)}, r_2^{(1)})$, $(t_1^{(2)}, r_1^{(2)})\}$.*

$S(\mathcal{T})$	$t_1^{(1)}$	$r_1^{(1)}$	$t_2^{(1)}$	$r_2^{(1)}$	$t_1^{(2)}$	$r_1^{(2)}$	$t_3^{(1)}$
σ	$t_1 \cdot OK$	$r_1 \cdot KO$	$t_2 \cdot OK$	$r_2 \cdot OK$	$t_1 \cdot OK$	$r_1 \cdot OK$	$t_3 \cdot OK$

Table 3.9: Partial observation example

Definition 49 (Partial order observation specificity)**.** *Let $\mathcal{T}$ be a sequence of multi-set and let $(S(\mathcal{T}), \lhd_1)$ and $(S(\mathcal{T}), \lhd_2)$ be two partial order observations over the same multi-set $\mathcal{T}$. $\lhd_1$ is* less specific *than $\lhd_2$ iff $\lhd_1 \subset \lhd_2$.*

Definition 50 (Partial order observation union). *Let $(S(\mathcal{T}_1), \lhd_1)$ and $(S(\mathcal{T}_2), \lhd_2)$ be two partial order observations, their union $(S(\mathcal{T}_1), \lhd_1) \cup (S(\mathcal{T}_2), \lhd_2) = (S(\mathcal{T}_1) \cup S(\mathcal{T}_2), \lhd_1 \cup \lhd_2)$.*

Now consider CPN net system $\langle N, M \rangle$ and let M' be marking reachable from M by the means of a trace τ, we denote by $[\overrightarrow{\tau}]^{M}_{M'} = \{\tau' \in T^* | M \xrightarrow{\tau} M' \wedge \overrightarrow{\tau} = \overrightarrow{\tau'}\}$.

Definition 51 (Minimal partial ordered observation). *Let $\mathcal{T}$ be a multi-set of observations between two markings M and M', the* minimum partial ordered observation *between M and M' is defined as $\lhd_{min}^{M \xrightarrow{\mathcal{T}} M'} : \mathcal{T} \times \mathcal{T}$ s.t. $\forall \lhd'$, which has the same observable transition sequence with $\lhd$, $\lhd$ is less specific than $\lhd'$.*

3-d-p example 12. *Suppose a principle of "taking first and then putting down the forks" and the activities of each philosopher are collected by different monitoring components, and given several partially ordered observations $(S(\mathcal{T}_1), \lhd_1) = (\{t_1^{(1)}, r_1^{(1)}, t_1^{(2)}, r_1^{(2)}\}, \{(t_1^{(1)}, r_1^{(1)}), (t_2^{(1)}, r_2^{(1)})\})$, $(S(\mathcal{T}_2), \lhd_2) = (\{t_2^{(1)}, r_2^{(1)}\}, \{(t_2^{(1)}, r_2^{(1)})\})$, and $(S(\mathcal{T}_3), \lhd_3) = (\{t_3^{(1)}\}, \{\emptyset\}$ with the global initial and final markings as M and M', the minimum partial ordered observation of the global system $\lhd_{min}^{M \xrightarrow{\mathcal{T}} M'} = \lhd_1 \cup \lhd_2 \cup \lhd_3$.*

Minimal partial order observation (w.r.t a multi-set $\mathcal{T}$ and two marking M and M') stores the minimum information on the order between occurrences for all traces with the same characteristic vector and confluent between a source and a target marking.

In this work, we suppose the CPN fault model only has the minimum of information about the order of transition (not modes).

Definition 52 (Minimal partial order observation function). *Given a CPN fault model N, we define observation function $\mathcal{O}bs : (\mathbb{M}_N \times \mathbb{M}_N) \times \mathcal{M}(T) \to (T \times \mathbb{N}) \times (T \times \mathbb{N})$ with $\mathcal{O}bs((M, M'), \mathcal{T}) = (S(T), \lhd^{M \xrightarrow{\mathcal{T}} M'})$ if $\exists \tau \in T^*$ with $M \xrightarrow{\tau} M' \wedge \overrightarrow{\tau} = \mathcal{T}$ and undefined otherwise.*

3.4 Related works

There are several works which use CPN as the model for diagnose. In most works, the colored tokens are normally used to model the system descriptions, so the places represent the faulty states and the diagnosis task is to reconstruct the possible paths which are consistent with the ordered observations.

[19] defined a CPN model, the color set includes an integer represents the severity level as the guard (the firing conditions for transitions) and a token identity represents control of system. The correct and faulty states are represented as places. The MBD diagnosis is formulated by hidden state history reconstruction, from event (e.g. alarm) observations. The paper modeled the asynchronous system and the on-the-fly diagnosis result is retrieved with CPN unfolding method.

[21] concerned the diagnosis problem caused by the change of system structure (e.g. add or delete components). So the dynamic components are represented as tokens and alarm are represented as the new tokens emitted by a transition through the ”reading arcs” (arcs do not consume/product any token). An unfolding approach is proposed for the CPN model by inferring the causal dependencies between the observed events, which means by eliminating the conflict events during unfolding the CPN. So the diagnosis problem can be expressed as the computation of an unfolding constrained by the observations, in order to retain the trajectories that explain the observations.

[127] proposed an intelligent event-oriented diagnosis methodology and diagnostic system architecture. The system descriptions (variables) are defined as the token colors (multi-sets). The faulty events are represented as the unobservable transitions in CPN model. The diagnosing approach is to construct the possible event sequences according the observations. The preconditions of the event set are described in a *rule table*, and the effect and consequences of the event itself is encoded in a *change table*. The dynamic model of the system is assumed to be partially unknown and it is refined using the observed event sequences by a learning method. The real-time diagnosis operates on the CPN model of the system and on the expected operating procedures comparing observed event sequences to the model-based prediction.

[89, 122] describe the modeling and use of CPN model for fault diagnosis and recovery respectively in power system and embedded control systems. The CPN model has the complex token types based on sets or complex sets containing the structured information for error handling. The diagnosis is achieved by marking graph reachability approach.

Chapter 4

CPN diagnosis based on inequations system

Given a DES, the diagnosis is to compare the observed behavior of the real system and the simulated behavior of its abstract model to detect, isolate, and explain the exceptions ([49]). So to find an adequate abstract model which can simulate and represent the normal and abnormal behaviors of the system is a fundamental step. In chapter 3, we have formally defined a CPN fault model which includes the data and transitions faults. The observations is considered as a trace of the transitions without mode. The status of the system during the evolution is represented as the markings. In this chapter, based on the CPN model and the observed trace, we study how to retrieve the faults to explain the observed symptoms in an algebraic way. The diagnosis approach is based on the reachability property of the CPN model, and realized by backward inferring.

4.1 Diagnosis problem

When a fault occurs at some moment, an exception or an alarm is observed, what we call in diagnosis literature, a symptom. Symptoms are presented due to some inconsistency concerns either the I/O interfaces or control flow faults. Those symptoms, in our CPN fault model, are represented as red tokens. When symptom(s) are detected by the monitoring component(s), the software system can continue running, and the diagnoser(s) are triggered. So our diagnosis approach can run online but when cooperating with the repairing components, it need to be stopped for performing possible repairing actions, which is required in the WSDIAMOND [138] project.

In some cases, some data is declaimed to be correct, thus the corresponding token colors are

black. And for the data that no correct or fault information is reported, the corresponding tokens colors stay unknown. So a symptom can be represented as a marking where the faulty data (or flow control) is marked as red tokens and the others can be marked either as black or unknown ones.

Definition 53 (CPN symptom markings)**.** *A marking M is a* symptom (exception) marking *iff* $\exists p$, $M(p)(r) \neq 0$[1]*, which is denoted as* $\hat{M}$.

Given a CPN model, a diagnosis problem is a 3-tuple of an initial marking, an observed (partially ordered) transitions trace, and a symptom marking, which contains the reported exception(s): the red (r) token(s), the possible correct data/control(s): the black (b) token(s) and the status unknown data/control(s): the $*$ token(s).

Definition 54 (CPN diagnosis problem)**.** *Given a CPN graph N, a diagnosis problem for N is a tuple* $\mathcal{D} = \langle M_0, (S(\mathcal{T}), \lhd), \hat{M} \rangle$*:*

- M_0 *is an initial marking of a CPN system;*
- $(S(\mathcal{T}), \lhd)$ *is a partially ordered observation;*
- $\hat{M}$ *is a symptom marking of a CPN system.*

In real life, only the detected faulty places are marked as a red token in a symptom marking, and the places, of which the token color are unknown, are marked as $*$. We say a marking which contains the color unknown tokens "covers" the markings that contain the color known tokens in all the same places. Thus a covering relation is introduced as follows:

Definition 55 (Covering relation)**.** *A covering relation $\succcurlyeq$ between colors $status = \{r, b, *\}$ ($status \in \Sigma$ is a color set) is a reflective, transitive, but not symmetric relation where any color covers itself and the $*$ color covers all colors (i.e $\succcurlyeq = \{(r,r), (b,b), (*,*), (*,r), (*,b)\}$). We extend the color covering relation to the multisets and markings as follows:*

- *let* $m, m' \in \mathcal{M}^+(\Sigma)$*, we have* $m' \succcurlyeq m$ *iff* $\sum_{c\in\Sigma} m(c) = \sum_{c\in\Sigma} m'(c) \wedge \forall c \neq *,\ m'(c) > 0 \Rightarrow m(c) \geqslant m'(c)$
- *let* M, M' *be two markings, we have* $M' \succcurlyeq M$ *iff* $\forall p \in P$, $M'(p) \succcurlyeq M(p)$

3-d-p example 13. *Given* $m_1 = 2`* + 2`b$*,* $m_2 = 1`* + 1`r + 2`b$ *and* $m_3 = 2`* + 1`r + 1`b$*, we have* $m_1 \succcurlyeq m_2$ *but no* $m_1 \succcurlyeq m_3$.

[1]Remember $M(p)$ is the color of place p in marking M (see definition 37 in chapter 3)

There is a causality relation between the initial and symptom markings ($M_0 + C \times \overrightarrow{\sigma} = M_n$, see section 3.2.2 in chapter 3). While in symptom marking $\hat{M}$, the correct/fault status information is less than the real final marking M_n. In the literature of the CPN diagnosis model, we say the symptom marking "covers" the final marking ($\hat{M} \succcurlyeq M_n$). By applying this "covering relation" to the incidence matrix equation of the CPN model, we get an inequations system (see equation 4.1).

$$\hat{M} \succcurlyeq M_0 + C \times \overrightarrow{\delta} \tag{4.1}$$

Consider our CPN fault model, intuitively, the faults can be either the faulty inputs or the KO modes of the transitions which are sufficient to reach the symptom marking through the observed trace. So a diagnosis solution should contain two parts: a subset of the place set P, of which the token colors are red in the initial marking; a subset of the transitions modes with the fault model KO: $FM^{-1}(KO)$ (see definitions 33 for CPN graph and 36 for fault model of CPN FM in chapter 3). We give now a definition of a diagnosis:

Definition 56 (CPN diagnosis)**.** *Let $\mathcal{D}$=$\langle M_0, (S(\mathcal{T}), \lhd), \hat{M}\rangle$ be a diagnosis problem for a CPN graph N, a diagnosis is defined as $Diag(\mathcal{D}) \subseteq 2^{FM^{-1}(KO) \cup P}$ s.t. $\forall Sol \in Diag(\mathcal{D})$, we have $\exists \delta \in Mod^*$ and $\exists M_0 \succcurlyeq M_0'$, s.t.:*

(i) $\forall m \in Sol$, $\overrightarrow{\delta}(m) \neq 0 \wedge \overrightarrow{\delta}_T = \mathcal{T} \wedge \lhd \subseteq \delta_T$;

(ii) $\forall p \in Sol$, $M_0'(p) = r \wedge \hat{M} \succcurlyeq M_0' + C \times \overrightarrow{\delta}$.

(i) $\overrightarrow{\delta}_T = \mathcal{T}$ defines the limit of the occurrence for each observed transition which can behave in different modes. $\lhd \subseteq \delta_T$ represents the observed partial ordered ($\lhd$) trace should be consistent with the transition trace δ_T of the modes trace δ.

(ii) indicates we are looking for a possible initial marking M_0' with $M_0' \succcurlyeq M_0$ and the inequations system (see equation 4.1) still holds.

Note that once transition t behaves under KO mode ($\overrightarrow{\delta}(m) > 0$), it should be included in diagnosis. So in this case, the real value of $\overrightarrow{\delta}(m)$ is not important for diagnosis (i.e., given $m = t_1 \cdot KO$, no matter $\overrightarrow{\delta}(t_1 \cdot KO) = 1$ or $\overrightarrow{\delta}(t_1 \cdot KO) = 2$, $t_1 \in Diag(\mathcal{D})$).

Definition 57 (CPN minimal diagnosis)**.** *Let $\mathcal{D}$=$\langle M_0, (S(\mathcal{T}), \lhd), \hat{M}\rangle$ be a diagnosis problem for a CPN model N, $\forall Sol \in Diag(\mathcal{D})$, Sol is minimal iff $\forall Sol' \subset Sol$, $Sol' \notin Diag(\mathcal{D})$. $Diag$ is minimal iff $\forall Sol \in Diag(\mathcal{D})$, Sol is minimal.*

p1	p2	p3	p11	p21	p22	p32	p13	p33
⟨ ∗	∗	∗	0	0	0	0	0	0 ⟩

Table 4.1: Dining philosophers: an initial marking M_0

3-d-p example 14. *Assume a scenario that each fork is in one plate, but we have no idea of their positions which means the initial marking M_0 contains only the unknown tokens (see table 4.1).*

Then the philosophers 1, 2, and 3 ate in order; then the philosopher 3 took the forks and found the fork series number in his right hand (suppose to be 3) is correct but the fork series number in his left hand (suppose to be 1) is wrong. We have no idea about the series number of the third fork. So we get a symptom marking $\hat{M}$ (see table 4.2) in which the place p_{33} contains a black token, p_{13} contains a red token, and p_2 contains an unknown token. Meanwhile a transition trace $\delta_T = t_1 r_1 t_2 r_2 t_3$ is observed by the monitoring component, and the corresponding partially ordered observation is denoted as $(\mathcal{T}_1, \lhd_1)$ with $\mathcal{T}_1 = \{1`t_1,\ 1`r_1,\ 1`t_2,\ 1`r_2,\ 1`t_3\}$ and $\lhd_1 = \{(t_1, r_1), (t_2, r_2)\}$. The corresponding characteristic vector $\overrightarrow{\delta}$ and the transition characteristic vector $\overrightarrow{\delta}_T$ are illustrated in tables 4.3 and 4.4. Note that the sum of the occurrence of modes OK and KO of transition r_1 is 1 and the sum of the occurrence of modes OK and KO of transition r_2 is 1.

p1	p2	p3	p11	p21	p22	p32	p13	p33
⟨ 0	∗	0	0	0	0	0	r	b ⟩

Table 4.2: Dining philosopher: a symptom marking $\hat{M}$

$t_1 \cdot OK$	$r_1 \cdot OK$	$r_1 \cdot KO$	$t_2 \cdot OK$	$r_2 \cdot OK$	$r_2 \cdot KO$	$t_3 \cdot OK$	$r_3 \cdot OK$
⟨ 1	n1	n2	1	n3	n4	1	0 ⟩

Table 4.3: Dining philosophers: a characteristic vector $\overrightarrow{\delta}$ with the transitions occurrence constraints $n_1 + n_2 = 1$ and $n_3 + n_4 = 1$

As an intuition, each diagnosis solution Sol of the diagnosis $Diag(\mathcal{D})$ is a subset of $\{p_1,\ p_2,\ p_3,\ md(r_1),\ md(r_2)\}$ which includes all the places that are initially marked and all the transitions modes which can behave in KO modes. We first give the answer to the diagnosis problem $\mathcal{D}$ that the diagnosis $Diag(\mathcal{D})$ should be $\{\{p_1\},\ \{md(r_1)\}\}$ which means there are two solutions to explain the diagnosis problem $\mathcal{D}$: either the fork number in the place p_1 is faulty ($\{p_1\}$), or the philosopher 1 has done something wrong ($\{md(r_1)\}$, means he did the release activity wrong). More specifically, for diagnosis solution $\{p_1\}$, concerning a diagnosis solution Sol (for example $Sol = \{p_1, p_2, md(r_1)\}$, which satisfies $\{p_1\} \subseteq Sol \subseteq \{p_1, p_2, p_3, md(r_1)\}$, we have: for any

t_1	r_1	t_2	r_2	t_3	r_3
$\langle 1$	1	1	1	1	$0 \rangle$

Table 4.4: Dining philosophers: a transition characteristic vector $\overrightarrow{\delta}_T$

possible mode sequence δ which is consistent with the partially observed trace $(S(\mathcal{T}_1), \lhd_1)$ and for all the initial markings $M_0 \succcurlyeq M_0'$.

- *$\forall m \in Sol$, $\overrightarrow{\delta}(m) \neq 0$ ($md(r_1) = 1$), and $\overrightarrow{\delta}_T = \mathcal{T}_1$ and $\lhd_1 \subseteq \delta_T$;*
- *$\forall p \in Sol$, $M_0'(p) = r$, s.t., $\hat{M} \succcurlyeq M_0' + C \times \overrightarrow{\delta}$, which, in this example, is $M_0'(p_1) = r$ and $M_0'(p_2) = r$;*

And similar for the diagnosis $\{md(r_1)\}$, any Sol, that satisfies the conditions $\{md(r_1)\} \subseteq Sol \subseteq \{p_1, p_2, p_3, md(r_1), md(r_2)\}$, satisfies the definition of the CPN diagnosis (see definition 56).

In the next section, we explain how to solve the diagnosis problem by satisfying these two conditions.

4.2 Diagnosis of CPN by inequations system solving

Given a CPN diagnosis problem $\mathcal{D} = \langle M_0, (S(\mathcal{T}), \lhd), \hat{M} \rangle$, a solution (see the part inside the dashed line frame in figure 4.1) for $\mathcal{D}$ is to check if there exists a $M_0' \in \mathbb{M}_N$, a $\delta \in Mod'$, with $M_0'[\delta\rangle M'$, s.t.: (i) $M_0 \succcurlyeq M_0'$; (ii) $\hat{M} \succcurlyeq M'$; (iii) $S(\mathcal{T}) = S(\overrightarrow{\delta_T})$

This means that we seek for an initial marking, which is covered by M_0, and a trace of transitions that leads to a marking M' that is covered by the symptom $\hat{M}$. The solution is the set of places, which appear in M_0 as $*$ and in M_0' as r, and the set of all transitions t_i that behave under faulty mode and appear in the run. So Sol of $\mathcal{D}$ is the set of all the possible solutions explained above, which is also the solutions of the inequations system 4.1. To solve inequations system, which contains a set of constraints of the transitions occurrence, is similar to solve Constraint Satisfaction Problems (CSP, [130]). While besides the transition modes occurrence constraints, the multi-set of enumerate token colors, data dependency functions and covering relations are not defined in the CSP solvers, so in this section, we propose the CSP-like diagnosis algorithms to solve the CPN diagnosis problems.

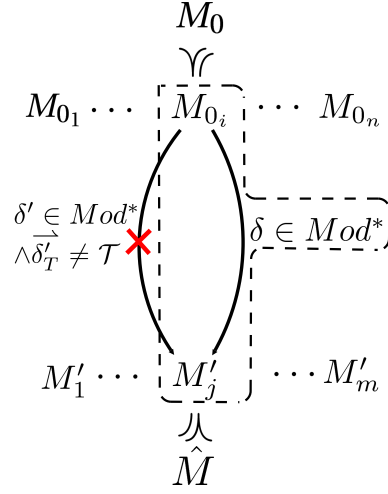


Figure 4.1: A solution for a CPN diagnosis problem, which consists of an initial marking M_{O_i} and a trace δ. M_{O_i} and δ' cannot be a solution because $\overrightarrow{\delta'_T} \neq \mathcal{T}$

4.2.1 Inequations system

According to the definition of diagnosis for CPN model (see definition 56), a diagnosis problem is transferred to a inequations system solving problem (see equation 4.2).

$$Q_{\hat{M}} \begin{cases} OC_{t_j} : \sum\limits_{m \in \overrightarrow{\delta}(md(t_j))} m = \overrightarrow{\delta}(t_j) \\ \cdots \\ Eq_{p_1} : \hat{M}(p_1) \succcurlyeq M_0(p_1) + C(p_1, .)\overrightarrow{\delta} \\ \cdots \\ Eq_{p_i} : \hat{M}(p_i) \succcurlyeq M_0(p_i) + C(p_i, .)\overrightarrow{\delta} \\ \cdots \\ Eq_{p_n} : \hat{M}(p_n) \succcurlyeq M_0(p_n) + C(p_n, .)\overrightarrow{\delta} \end{cases} \tag{4.2}$$

$n = |P|$ represents the number of the places. If a transition t is allowed to behave either in OK or KO mode, an occurrence constraint equations OC_t is associated to make sure the sum of the occurrence of OK and KO modes is equal to the sum of the occurrence of transition t in the observed trace. So in fact OC_t contains two variables which represent the modes occurrence of OK and KO of transition t.

Each place p corresponds to an inequation Eq_p where the left part is $l(Eq_p)$=$\hat{M}(p)$ and the right part is $r(Eq_p)$=$M_0(p) + C(p, .) \times \overrightarrow{\delta}$. Each $r(Eq_p)$ consists of the items which have the similar

structures: $item_i = \overrightarrow{\delta}(m_i) \times \Psi(cd(p'))$ with $p' \in P$ (Note that $M_0(p)$ can be seen as $1`M_0(p)$). The occurrence $\overrightarrow{\delta}(m_i)$ can be a constant n_j or a variable v_j, and the color expression $\Psi(cd(p'))$ can be a constant color r, b, $*$, a variable $\chi_{p'}$, which represents the color of place p', or a data dependency function which can be $FW(\chi_{p'})$, $EL(\chi_{p'_1}, \cdots, \chi_{p'_n})$, or $SRC(t)$. Denote $r(Eq_p)^+$ as the not negative items on the right side of the inequation Eq_p.

3-d-p example 15. *Concerning the diagnosis problem, an inequations systems, which is got by matrix calculation (see table 4.5), should be satisfied.*

$\hat{M} \curlyeqsucc M_0+$

C	t_1	t_2	t_3	r_1		r_2		r_3
				OK	KO	OK	KO	
p_1	$-\chi_{p_1}$		$-\chi_{p_1}$	$FW(\chi_{p_{11}})$	r			$FW(\chi_{p_{13}})$
p_2	$-\chi_{p_2}$	$-\chi_{p_2}$		$FW(\chi_{p_{21}})$	r	$FW(\chi_{p_{22}})$	r	
p_3		$-\chi_{p_3}$	$-\chi_{p_3}$			$FW(\chi_{p_{32}})$	r	$FW(\chi_{p_{33}})$
p_{11}	$FW(\chi_{p_1})$			$-\chi_{p_{11}}$	$-\chi_{p_{11}}$			
p_{21}	$FW(\chi_{p_2})$			$-\chi_{p_{21}}$	$-\chi_{p_{21}}$			
p_{13}			$FW(\chi_{p_1})$					
p_{33}			$FW(\chi_{p_3})$					
p_{22}		$FW(\chi_{p_2})$				$-\chi_{p_{22}}$	$-\chi_{p_{22}}$	
p_{32}		$FW(\chi_{p_3})$				$-\chi_{p_{32}}$	$-\chi_{p_{32}}$	

$\times$

t_1:	1
$r_1.OK$:	n_1
$r_1.KO$:	n_2
t_2:	1
$r_2.OK$:	n_3
$r_2.KO$:	n_4
t_3:	1
r_3:	0

Table 4.5: Dining philosopher: inequations system in form of matrix calculation

The inequations system of the three dining philosophers is shown in equation 4.3. And for example, in an inequation Eq_{p_1}, the items on the right side are: $*$, $-1`\chi_{p_1}$, $-1`\chi_{p_1}$, $n_1`FW(\chi_{11})$, and $n_2`r$. $r(Eq_p)^+$ includes the items $*$, $n_1`FW(\chi_{11})$, and $n_2`r$.

$$\begin{cases} OC_{r_1} : n_1 + n_2 = 1 \\ OC_{r_2} : n_3 + n_4 = 1 \\ Eq_{p_1} : 0 \curlyeqsucc * - 1`\chi_{p_1} - 1`\chi_{p_1} + n_1`FW(\chi_{11}) + n_2`r \\ Eq_{p_2} : * \curlyeqsucc * - 1`\chi_{p_2} - 1`\chi_{p_2} + n_1`FW(\chi_{p_{21}}) + n_2`r + n_3`FW(\chi_{p_{22}}) + \\ n_4`r \\ Eq_{p_3} : 0 \curlyeqsucc * - 1`\chi_{p_3} - 1`\chi_{p_3} + n_3`FW(\chi_{p_{32}}) + n_4`r \\ Eq_{p_{11}} : 0 \curlyeqsucc 0 + 1`FW(\chi_{p_1}) - n_1`\chi_{p_{11}} - n_2`\chi_{p_{11}} \\ Eq_{p_{21}} : 0 \curlyeqsucc 0 + 1`FW(\chi_{p_2}) - n_1`\chi_{p_{21}} - n_2`\chi_{p_{21}} \\ Eq_{p_{13}} : r \succcurlyeq 0 + 1`FW(\chi_{p_1}) \\ Eq_{p_{33}} : b \succcurlyeq 0 + 1`FW(\chi_{p_3}) \\ Eq_{p_{22}} : 0 \succcurlyeq 0 + 1`FW(\chi_{p_2}) - n_3`\chi_{p_{22}} - n_4`\chi_{p_{22}} \\ Eq_{p_{32}} : 0 \succcurlyeq 0 + 1`FW(\chi_{p_3}) - n_3`\chi_{p_{32}} - n_4`\chi_{p_{32}} \end{cases} \tag{4.3}$$

According to the token colors in the symptom marking $\hat{M}$, the inequations $Q_{\hat{M}}$ is divide into

three exclusive subsets: $\hat{M} = Q^r_{\hat{M}} \cup Q^b_{\hat{M}} \cup Q^*_{\hat{M}}$ which are defines as follows:

$$Q^r_{\hat{M}} = \{Eq_p | l(Eq_p) = r\} \tag{4.4}$$

$$Q^b_{\hat{M}} = \{Eq_p | l(Eq_p) = b\} \tag{4.5}$$

$$Q^*_{\hat{M}} = \{Eq_p | l(Eq_p) = * \vee l(Eq_p) = 0\} \tag{4.6}$$

$Q^r_{\hat{M}}$ represents an inequations set in which the left side of the inequation is a red (r) token; $Q^b_{\hat{M}}$ represents the inequations set in which the left side of the inequation is a black (b) token; $Q^*_{\hat{M}}$ represents the inequations set in which the left side of the inequation is an unknown ($*$) token or no token.

3-d-p example 16. *The inequations system (see equation 4.3) can be divided as:* $Q^b_{\hat{M}} = \{Eq_{p_{33}}\}$, $Q^r_{\hat{M}} = \{Eq_{p_{13}}\}$, *and* $Q^*_{\hat{M}} = \{Eq_{p_1}, Eq_{p_2}, Eq_{p_3}, Eq_{p_{11}}, Eq_{p_{21}}, Eq_{p_{22}}, Eq_{p_{32}}\}$.

In the MBD theory, the more information is considered in the symptom, the more precisely the diagnosis is. In the CPN model for diagnosis, the black token(s) in the symptom marking is helpful to reduce the number of the possible diagnosis results. What we do is to integrate this "black token(s)" information into other inequations to form an updated inequations system.

4.2.2 Algorithms

To solve the inequations system, there are two steps to follow:

(i) to start from the inequations of $Q^b_{\hat{M}}$, to infer the possible related black toke(s) and transition(s) behave under OK mode, and to update the inequations system if necessary (function $getImpossibleSols$, see algorithm 2);

(ii) to start from the inequations of $Q^r_{\hat{M}}$ and to infer the diagnosis in the set of $Q^*_{\hat{M}}$ without violate the transition occurrence constraints (function $getDiag$, see algorithm 4).

(i) The inferring process starts from the left side of an inequation of $Q^b_{\hat{M}}$. To make sure the black token "covers" the color expressions on the right side, it is obliged to make sure that on the right side of the inequation, there is at least one positive b. The function $getImpossibleSols$ has the following sketch: to solve the inequations system by recursively calling function $inferOneBlack$, which solves one inequation of $Q^b_{\hat{M}}$, backward along the data dependency functions.

(ii) To make sure the red token on the left side ”covers” the expressions on the right side, it sufficient to have at least one r token on the right side. That is, all the combinations of the value of the items $\overrightarrow{\delta}(m_i) \times \Psi(cd(p'))$, which has at least one r, are sufficient to propagate a r token on the left side. So the inferring result of each inequation can be represented as a set of places that ”might” contain a red token, and a set of transitions modes that ”might” be KO. The function $getDiag$ has the following sketch: to solve the inequations system by recursively calling function $inferOneRed$, which solves one inequation of $Q^r_{\hat{M}}$, backward along the data dependency functions.

$getImpossibleSols$ **function**

To solve one inequation of $Q^b_{\hat{M}}$ (by function $inferOneBlack$), there are two kinds of inequations to consider:

(i) the inequations have no negative items, as the r tokens cannot be consumed, the value of each the nonnegative item should be b. So the related occurrence constraints can be updated and the tokens it depends on in the data dependency functions should be b.

(ii) the inequations have negative items, so there are many possible combinations of the value of the unknown variables. And there is only one impossible situation: its initial marking is r, and all its input transitions behave each time under KO mode. This information can help to reduce the diagnosis result only if all the following conditions are satisfied: its initial marking is known to be r, each of its input transitions is fired only once, and there is a diagnosis solution which says all its input transitions behave under KO mode. In this case, this solution should be deleted from the diagnosis (see example 17).

3-d-p example 17. *Given an inequation* $Eq_{p_1} : b \succcurlyeq r - 2^{\backprime}\chi_{p_1} + n_1^{\backprime}FW(\chi_{p_2}) + n_2^{\backprime}r + n_3^{\backprime}FW(\chi_{p_3}) + n_4^{\backprime}r$*, and occurrence constraints* $n_1 + n_2 = 1$ *and d* $n_3 + n_4 = 1$ *if there is a solution* $sol = \{p_2, p_3\}$ *in the diagnosis result, we should delete it.*

(i) For the each item of $r(Eq_p)$, there are two cases: (see algorithm 1):

- $v_j^{\backprime}r$ (line 2), then the occurrence variable v_j must be 0;
- $v_j^{\backprime}\chi_p$ or $v_j^{\backprime}func$ with $func$ is a data dependency function (line 3), if $v_j > 0$, then the color variable (including the variables in $func$), χ_p must be b;

3-d-p example 18. *Consider the inequation $Eq_{p_{33}} \in Q^b_{\hat{M}}$, to apply the function $getImpossibleSols(Eq_{p_{33}})$, there are two no negative items: 0 and $1`FW(\chi_{p_3})$, so the only possible to propagate a b token on the left side is that $FW(\chi_{p_3}) = b$ (the third case), which infers $\chi_{p_3} = b$.*

Algorithm 1 $inferOneBlack(Eq_p)$: solve one $Q^b_{\hat{M}}$ inequation to get a constraint

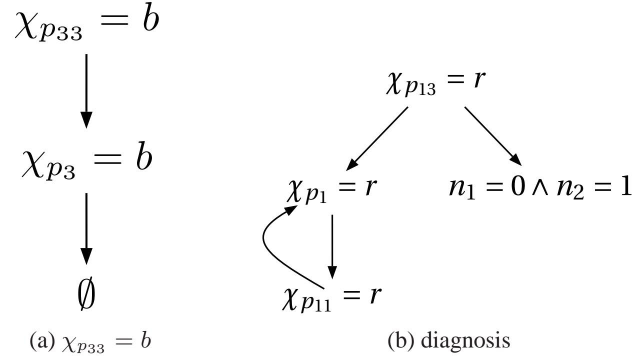
Input: Eq_p: one $Q^b_{\hat{M}}$ inequation concerns a place p;
1: **if** $r(Eq_p)$ has no negative item **then**
2: update OC with condition $t_i.KO = 0$;
3: **return** all color variables $\chi_{p'}$ (including the variables in the data dependency functions) for further inferring;
4: **else if** $r(Eq_p)$ has negative items **then**
5: **if** $M_0(p) = r$ and all the related transitions occurrences are constrained to be 1 **then**
6: **return** all the KO modes of which transitions in Eq_p as an impossible solution;
7: **end if**
8: **end if**

(ii) To retrieve the impossible solutions set for one inequation in $Q^b_{\hat{M}}$, the idea is to start from each black token in $\hat{M}$, and infer backward recursively (line 8) through all the color variables. In algorithm 2, two kinds of the inequations during the further solving are considered:

- to further solve a inequation $Eq_{p'} \in Q^*_{\hat{M}}$ with the left part is $*$ (line 8), $Eq_{p'}$ is transformed into a new inequation in $Q^b_{\hat{M}}$ by evaluating the $*$ on the left side of the inequation as b;
- to further solve a inequation $Eq_{p'} \in Q^*_{\hat{M}}$ with the left part is 0 (line 10), $Eq_{p'}$ is transformed into a new inequation in $Q^b_{\hat{M}}$ by adding a color variable $\chi_{p'}$ on both sides of the inequation and evaluating the left one as b.

Thus a new inequations system is constructed for further constraint solving and the "covering relation" of the inequation $Eq_{p'}$ is not violated.

3-d-p example 19. *As a continue of the example, function $getImpossibleSols$ starts from $Eq_{p_{33}}$, which calls the function $inferOneBlack(Eq_{p_{33}})$ and leads to $\chi_{p_3} = b$. Then the recursion starts. As $l(Eq_{p_3}) = 0$, we add χ_{p_3} on both sides of Eq_{p_3} and evaluate the one on the left side as b (we denote the new inequation as Eq'_{p_3}). But $M_0(p_3) = *$, there is no impossible solution for Eq'_{p_3} (see figure 4.2(a)).*



(a) $\chi_{p_{33}} = b$ (b) diagnosis

Figure 4.2: Dining philosopher: apply function $getImpossibleSols$ (algorithm 2) or $getDiag$ (algorithm 4) as a processes top-down, which means to search backward along the data dependency functions, and stops when reaching stable status.

Algorithm 2 $getImpossibleSols(Q_{\hat{M}})$: get the impossible solutions for $Q_{\hat{M}}$

Input: $Q_{\hat{M}}$: the inequations system;
1: $ImpossibleSols = \emptyset$ as the set of the impossible solutions;
2: **ForEach** $Eq_p \in \hat{M}^b$ **do**
3: apply algo 1 and store the returns in $ImpossibleSols$;
4: denote $Q^{b'}_{\hat{M}}$ as the set of the inequations need to solve further;
5: **end for**
6: **ForEach** $Eq_p \in Q^{b'}_{\hat{M}}$ **do**
7: **if** $l(Ep_p) = 0$ **then**
8: replace Eq_p with $b \succcurlyeq \chi_p + r(Eq_p)$;
9: **else if** $l(Ep_p) = *$ **then**
10: replace Eq_p with $b \succcurlyeq r(Eq_p)$;
11: **end if**
12: Recursion:
13: apply algo 1;
14: update $\hat{M}^*$ and $\hat{M}^b$;
15: **end for**
16: **return** $ImpossibleSols$;

During the diagnosis inferring, each impossible solution should be considered, so the impossible solutions set is the union of the impossible solutions for each inequation in $Q^b_{\hat{M}}$ (line 3 in algorithm 2).

Diagnosis inferring

The diagnosis algorithm executes backward reasoning recursively for each inequation $Eq_p \in Q^r_{\hat{M}}$ and then combines all the diagnosis results (algorithm 4). The impossible solutions set $ImpossibleSols$ should be considered to reduce the diagnosis result.

So the inferring principles of $inferOneRed(Eq_p)$ with $Eq_p \in Q^r_{\hat{M}}$ can be described as: for each positive item $item^+_j$ on the right side $r(Eq_p)$, its value *might* be r. In algorithm 3, two cases of the expression items are considered:

- $v_j`r$ (line 2), then the occurrence variable v_j must be larger than 0;
- $v_j`\chi_{p_j}$ or $v_j`func$ with $func$ is a data dependency function (line 3), if $v_j > 0$, then all the color variables χ_{p_j} (including the variables in $func$) might be r;

3-d-p example 20. *Consider the inequation $Eq_{p_{13}} \in Q^r_{\hat{M}}$, by applying the function $inferOneRed$ $(Eq_{p_{13}})$, there are two no negative items: 0 and $1`FW(\chi_{p_1})$. So the only possibility to propagate a r token on the left side is that $FW(\chi_{p_1}) = r$ (the third case), which infers $\chi_{p_1} = r$.*

Algorithm 3 $inferOneRed(Eq_p)$: partially solve a $Q^r_{\hat{M}}$ inequation

---

Input: Eq_p: one $Q^r_{\hat{M}}$ inequation concerns a place p;
Output: $Diag$: Diagnosis set;
1: **ForEach** positive item in $l(Eq_p)$ **do**
2: $Diag.add(m_{n_i})$ if $m_{n_i} \notin$;
3: $Diag.add(p')$ if $\chi_{p'}$ is a color variable (includes the variables in the data dependency functions);
4: **end for**
5: **return** $Diag$;

The part on the right side of an inequation is an expression composed by data dependency functions, constants, and the corresponding place variables which may have positive or negative occurrences. Solving the inequation consists in canceling the negative terms in the right part, keeping the positive color functions, and evaluating the positive occurrence n_i of red tokens (r) as $n_i > 0$ to explain the red token on the left side of the inequation (see algorithm 3). Algorithm 3 looks for, in one inequation, the possible diagnosis corresponding to one symptom place p in a symptom marking. And at the same time, it looks for the candidate inequations which can explain further how the red token is propagated. So to completely solve a red token in the symptom marking, a searching algorithm to recursively back reason by reconstructing $Q^r_{\hat{M}}$ and $Q^*_{\hat{M}}$ (algorithm 4).

To retrieve the diagnosis for one inequation in $Q^r_{\hat{M}}$, the idea is to recursively infer to other inequations for further solving through the data dependency functions and the color variables. In algorithm 4, two kinds of the inequations during the further solving are considered:

- to further solve a inequation $Eq_{p'}$ with the left part is 0 (line 5), $Eq_{p'}$ is transformed into a new inequation in $Q^r_{\hat{M}}$ by adding a color variable $\chi_{p'}$ on both sides of the inequation and evaluating the left one as r.
- to further solve a inequation $Eq_{p'}$ with the left part is $*$ (line 7), $Eq_{p'}$ is transformed into a new inequation in $Q^r_{\hat{M}}$ by evaluating the $*$ on the left side of the inequation as r;

Thus a new inequations system is constructed for further diagnosis solving and the "covering relation" of the inequation $Eq_{p'}$ is not violated.

Algorithm 4 $getDiag(Q_{\hat{M}})$: Diagnosis with constraints

Input: $Q_{\hat{M}}$: the inequations system;
Output: $Diag$: diagnosis set;

```
1:  Diag = ∅;
2:  ForEach Eq_p ∈ M̂^r do
3:      apply algo 3
4:      ForEach Eq_p ∈ M̂^* do
5:          if p ∈ Diag and l(Ep_p) = 0 then
6:              replace Eq_p with r ≽ χ_p + r(Eq_p);
7:          else if p ∈ Diag and l(Ep_p) = * then
8:              replace Eq_p with r ≽ r(Eq_p);
9:          end if
10:         Recursion:
11:                 apply algo 3;
12:                 update Q^*_M̂ and Q^r_M̂;
13:         merge the Diag and denote as Diag_p;
14:     end for
15: end for
16: Diag = ×∪ Diag_p \ ImpossibleSols;
17: return Diag;
```

3-d-p example 21. *As a continue of the example, function* $getDiag(Q''_{\hat{M}}$ *starts,* $\chi_{p_1} = r$ *call the function* $inferOneRed(Eq_{p_{13}})$*, and get* Eq_{p_1}*, which needs to solve further. And then function* $inferOneRed(Eq_{p_1})$ *is called in recursion. As* $l(Eq_{p_1}) = 0$*, we add* χ_{p_1} *on both sides of* Eq_{p_1} *and evaluate the one on the left side as* r *(we denote the new inequation as* Eq'_{p_1}*). Then we infer*

among the no negative items $*$, $1`\chi_{p_1}$ *(the new item added during the former step),* $n_1`\chi_{p_{11}}$, *and* $n_2`r$. *Meanwhile we should consider the occurrence constraint* $OC_{r_1} : n_1 + n_2 = 1$. *To make sure to propagate a* r *token on the left side of* Eq_{p_1}, *we have three possibilities:* $\chi_{p_1} = r$; $\chi_{p_{11}} = r \wedge n_1 = 1 \wedge n_2 = 0$ *(which means* r_1 *behaves in* OK *mode); or* $n_1 = 0 \wedge n_2 = 1$ *(which means* r_1 *behaves in* KO *mode). So up to now,* $DiagP = \{p_1, p_{11}\}$, *and* $DiagM = \{md(r_1)\}$. *Note that* $M_0(p_{11}) = 0$, *so* $Eq_{p_{11}}$ *need to solve further.*

Then we can construct a new inequations system $Q''_{\hat{M}}$ *with* $Q''^{b}_{\hat{M}} = \{Eq_{p_{33}}, Eq'_{p_3}, Eq'_{p_{32}}\}$, $Q''^{r}_{\hat{M}} = \{Eq_{p_{13}}, Eq'_{p_1}\}$, *and* $Q''^{*}_{\hat{M}} = \{Eq_{p_2}, Eq_{p_{21}}, Eq_{p_{22}}, Eq_{p_{11}}\}$.

Similarly, $Eq_{p_{11}}$ *is transformed to* $Eq'_{p_{11}}$ *by adding* $\chi_{p_{11}}$ *on both sides of* $Eq_{p_{11}}$ *and evaluating the one on the left side as* r. *Then we get the result* $Eq_{p_1} = r$ *(or* $DiagP = \{p_1\}$*). As there is no more inequation to infer, the function* $getDiag(Q''_{\hat{M}})$ *returns the result* $Diag = \{p_1, md(r_1)\}$. *The recursive process of calculating* $getDiag(Q_{\hat{M}})$ *by applying algorithm 4 is illustrated in figure 4.2(b).*

Multiple faults diagnosis

The diagnosis should be able to explain all the possibilities that how the symptom marking is generated. That is, each item in the diagnosis is a possible explanation for each red token in the symptom marking. Algorithm 4 (lines 1 to 15) calculate all the possible explanations for one red token. The union set of all the single faults (one red token) is a diagnosis solution for the inequations systems $Q_{\hat{M}}$. We use a *multi fault operator* $\overset{\cup}{\times}$ to integrate the solutions (line 16 in algorithm 4).

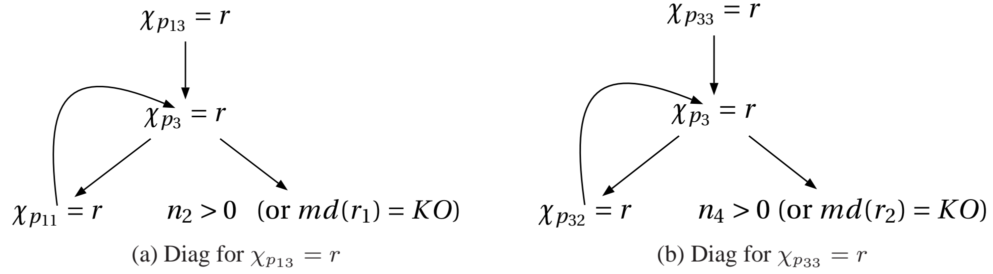
Definition 58 (Multi fault operator)**.** $\overset{\cup}{\times}$ *is an operator that calculates the Cartesian product and then keeps the minimal subsets.*

3-d-p example 22. *Still consider the same diagnosis problem, as it is a single fault diagnosis problem (*$Q^{r}_{\hat{M}} = \{Eq_{p_{13}}\}$ *which has only one inequation), we need not to use* $\overset{\cup}{\times}$ *operator to calculate the diagnosis for multiple faults. So the diagnosis* $Diag(\mathcal{D})$ *of the diagnosis problem* $\mathcal{D}$ *is* $\{p_1, md(r_1)\}$. *It means there are only two explanations: either the initial value of* p_1 *is faulty, or philosopher 1 has made a mistake when he released the forks.*

3-d-p example 23. *Now consider a multiple faults scenario, in which the symptom marking has red tokens in place* p_{13} *and* p_{33} *(see table 4.6). The initial marking and observed trace keeps unchanged (see tables 4.1 and 4.3). So the 3th philosopher finds faulty forks in both hands, which means* $\chi_{f_{13}} = r$ *and* $\chi_{f_{33}} = r$. *By applying the diagnosis algorithm, the diagnosis process is illustrated in figure 4.3.*

p1	p2	p3	p11	p21	p22	p32	p13	p33
0	*	0	0	0	0	0	r	r

Table 4.6: Dining philosopher: symptom marking $\hat{M}$ with multiple faults



(a) Diag for $\chi_{p_{13}} = r$ (b) Diag for $\chi_{p_{33}} = r$

Figure 4.3: Dining philosopher diagnosis: apply the diagnosis algorithm 4 as a processes top-down, which means to search backwards along the data dependency functions, and stop when arriving to a node which has been visited.

The diagnosis is show in the following equation (equation 4.7), which means 4 diagnosis solutions can explain the symptom marking: $\{p_1, p_3\}$*: the forks 1 and 3 were not in the right plates at the beginning;* $\{p_1, md(r_2)\}$*: fork 1 was not in the right plate and philosopher 2 made a mistake when he released the forks;* $\{md(r_1), p_3\}$*: philosopher 1 made a mistake when he released the forks and fork 3 was not in the right plate; and* $\{md(r_1), md(r_2)\}$*: philosophers 1 and 3 made mistakes when they released their forks.*

$$\begin{aligned} Diag(\mathcal{D}) &= \{\{p_1\}, \{md(r_1)\}\} \overset{\cup}{\times} \{\{p_3\}, \{md(r_2)\}\} \\ &= \{\{p_1, p_3\}, \{p_1, md(r_2)\}, \{md(r_1), p_3\}, \{md(r_1), md(r_2)\}\} \end{aligned} \tag{4.7}$$

4.3 The minimality of CPN diagnosis

The diagnosis algorithms perform the algebraic approach, which is effective and capable for handling the cyclic observation traces. But at the same time, the use of the characteristic vector loses part of the execution order of the transition. In this case, how to ensure the minimality of the diagnosis?

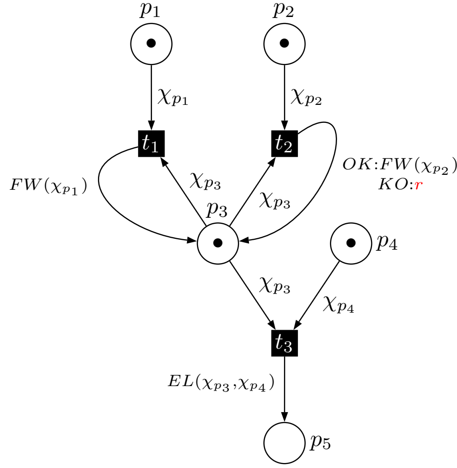


Figure 4.4: an example of CPN in which the execution order of transition t_1 and t_2 may effect the diagnosis result

C	t_1	t_2		t_3
		OK	KO	
p1	-χ_{p_1}			
p2		-χ_{p_2}	-χ_{p_2}	
p3	FW(χ_{p_1}) $-\chi_{p_3}$	FW(χ_{p_2}) $-\chi_{p_3}$	r-χ_{p_3}	-χ_{p_3}
p4				-χ_{p_4}
p5				EL(χ_{p_3},χ_{p_4})

Table 4.7: Incidence matrix of the example illustrated in figure 4.4

Assume an small CPN as illustrated in figure 4.4, we have its incidence matrix as in table 4.7 and the initial and symptom marking in table 4.8.

M_0	$\hat{M}$	$\overrightarrow{\sigma}$			
⟨p1,p2,p3,p4,p5⟩	⟨p1,p2,p3,p4,p5⟩	$t_1 \cdot OK$	$t_2 \cdot OK$	$t_2 \cdot KO$	$t_3 \cdot OK$
b , b , b , $*$, 0	0 , 0 , 0 , 0 , r	1	n1	n2	1

Table 4.8: Initial, symptom marking and characteristic vector of the example in figure 4.4

Then instanced inequations system is shown in equation 4.8.

$$\begin{cases} n_1 + n_2 = 1 \\ p_1 : 0 \succcurlyeq b - 1`\chi_{p_1} \\ p_2 : 0 \succcurlyeq b - n_1`\chi_{p_2} - n_2`\chi_{p_2} \\ p_3 : 0 \succcurlyeq * + 1`FW(\chi_{p_1}) - 1`\chi_{p_3} + n_1`FW(\chi_{p_2}) - n_1`\chi_{p_3} + n_2`r - n_2`\chi_{p_3} \\ -1`\chi_{p_3} \\ p_4 : 0 \succcurlyeq * - 1`\chi_{p_4} \\ p_5 : r \succcurlyeq 0 + 1`EL(\chi_{p_3}, \chi_{p_4}) \end{cases} \tag{4.8}$$

Suppose a completely ordered observation $(\{t_1, t_2, t_3\}, \{(t_2, t_1), (t_1, t_3)\})$, its minimal diagnosis should be $Diag(\mathcal{D}) = \{\{p_4\}\}$ if we compute with other diagnosis methods, like DES diagnoser or PN unfolding. In fact, $\{\{p_4\}, \{md(t_2)\}\}$ is the diagnosis for complete observation $(\{t_1, t_2, t_3\}$, $\{(t_1, t_2), (t_2, t_3)\})$ which generates the same symptom marking (in table 4.8).

In this CPN model, the execution order of transitions t_1, t_2 and t_3 is not defined, and t_1 and t_2 can be fired in parallels. So its partial order relations set $\lhd = \emptyset$. In this case, both the observations $(\{t_1, t_2, t_3\}, \{(t_1, t_2), (t_2, t_3)\})$ and $(\{t_1, 1vt_2, t_3\}, \{(t_2, t_1), (t_1, t_3)\})$ are legal. And they corresponds to a same characteristic vector as shown in table 4.8. By applying the algorithms 1-4 on the inequations system 4.8, the diagnosis result is: $Diag(\mathcal{D}) = \{\{p_4\}, \{md(t_2)\}\}$, which is a correct minimal diagnosis for the CPN model.

So the loss of the partial events order cannot make sure the retrieved diagnosis is minimal but is a *superset* of the minimal diagnosis. As the observation is defined as partially ordered, the impossible solutions for the observations is allowed to be minimal. In this case, the CPN model can fully models the parallel or concurrent events. This is very useful for modeling the complicate systems, like distributed systems (see chapter 6), of which observations are distributed also.

4.4 Related work

[144] proposes a decentralized model-based diagnosis algorithm based on the similar PNs model ([78]) by searching the possible trajectories backward. But in [144], local diagnoser does not support iteration of the system execution.

[118] models a modular interacting system as a set of place-bordered Petri nets and proposes a distributed online diagnosis which applies algebra calculations from the local models and the communicating messages between them. But the fact that [118] models the state of a model as a transition which causes the combinatorial explosion of the state space.

In [4], the similar definition of PN model for diagnosis are defined: the colored token, the covering relationships between different tokens. The diagnosis is retrieved by backward reachability search. Like [118], its simple Petri nets definition are too limited to deal with the data aspects

A similar diagnosis approach has been proposed in [5], of which we use the same data dependency relation. But [5] does not support loops in system process while we represent loops as the occurrence in a characteristic vector. In such way, we solve the loops without extra cost. The consistency-base diagnosis approach proposed in [5] is more abstract but loses the precision on the modeling level.

To generate each occurrence constraint equation, the diagnoser need to parse the observation trace to calculate the occurrence of each transition. Thus our diagnosis approach very slightly depends on the length of observation trace. During the diagnosis process, although our approach

contains recursive process, in worst case, each inequation is checked at most once. For each inequation concerning place p, the time of calculating depends on the number of the data dependency functions defined in scope of ${}^{\bullet}p$, which is $|Mod|$ in worst case, where $|Mod|$ is the sum of modes. So in the worst case, the time of the calculation of diagnosis is: $|P| \times (|Mod| + |T|)$, where $|P|$ is the number of places, $|T|$ is the number of transitions. As $|Mod| > |T|$, so the complexity of the diagnosis is $O(|P| \times |Mod|)$. But in the process which contain loops, the complexity of our algorithm is much less than those of [141], [5], [144] ($O(|P| \cdot |\delta|^2)$, where $|\delta|$ is the length of a trace).

Part II

Application: data fault diagnosis of decentralizes Web services

Chapter 5

Web services Application

5.1 Introduction

In this chapter we give an overview of the service-oriented architecture (SOA) based on Web services, and the related protocols of designing, composing, and executing Web services. A formal method is given to translate the standard Web service protocols (like structured programming languages) into CPN model for diagnosis. A foodshop case study example is introduced. The related works about the diagnosis approaches applied to Web services applications are discussed also.

5.2 SOA and Web service

The speeding increase of the information systems complexity makes it necessity to integrate the different systems within multiple business domains and offers seamless services to the clients, which is named as service-oriented architecture (SOA).

SOA defines how to integrate widely disparate applications for a world that is Web based which use multiple implementation platforms. Rather than defining an API, SOA defines the interface in terms of protocols and functionalities. An endpoint is the entry point for such an SOA implementation.

Web services can implement SOA by making functional building-blocks, which are accessible over standard Internet protocols, independent of platforms and programming languages.

A Web service can be basic or composite, and the basic ones make up composite ones. Normally, we construct composite Web services in two ways: the orchestrated [9] (figure 5.1a) or choreographed [61] (figure 5.1b), which differ in executability and control.

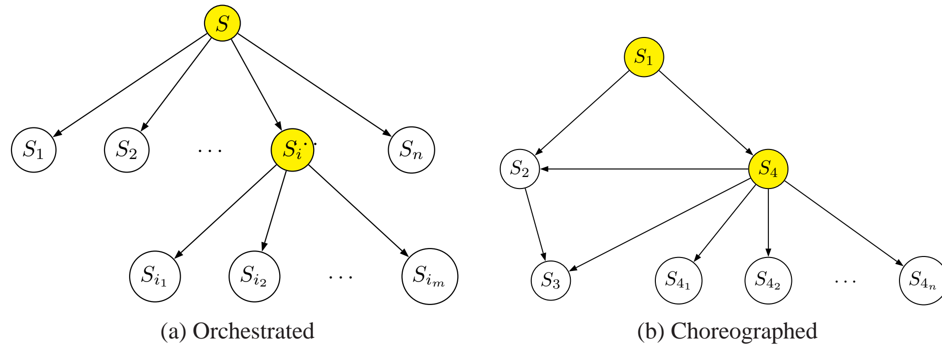


Figure 5.1: Composite Web service structure

An orchestrated Web service always represents control from one party¡¯s perspective. There is an orchestrating service which is in charge of communicate with all other service to perform a process. The orchestration can be recursive so a set of orchestrated Web services can a tree-shape organization structure.

Choreography ”tracks the message sequences among multiple parties and sources-typically the public message exchanges that occur between Web services-rather than a specific business process that a single party executes” [95]. So there is no central role for a set of choreographed Web services.

Choreography specifies a protocol for peer-to-peer interactions, defining, e.g., the legal sequences of messages exchanged with the purpose of guaranteeing interoperability. Such a protocol is not directly executable, as it allows many different realizations.

There are three roles related to the Web services consummation: the requester (or clients), services themselves, the inventory (registry). These three roles communicate between each other through different protocols: UDDI, SOAP, WSDL. Figure 5.2 illustrates the relationships between Web service related roles and the communication protocols.

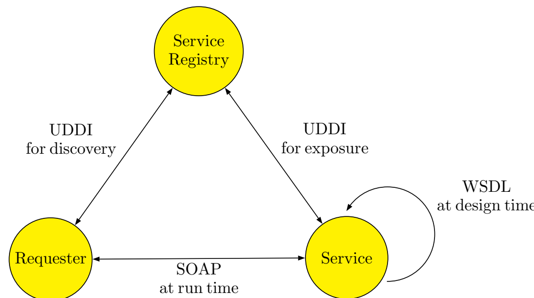


Figure 5.2: The relations between the roles and protocols

5.2.1 SOAP

Simple Object Access Protocol (SOAP) [43] is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on XML as its message format, and usually relies on other Application Layer protocols (most notably Remote Procedure Call (RPC) and HTTP) for message negotiation and transmission. SOAP can form the foundation layer of a web services protocol stack, providing a basic messaging framework upon which web services can be built.

SOAP consists of four parts:

1. The SOAP envelope construct defines an overall framework for expressing what is in a message, which should deal with it, and whether it is optional or mandatory.

2. The SOAP encoding rules define a serialization mechanism that can be used to exchange instances of application-defined data types.

3. The SOAP RPC representation defines a convention that can be used to represent remote procedure calls and responses.

4. The SOAP binding defines a convention for exchanging SOAP envelopes between peers using an underlying protocol for transport.

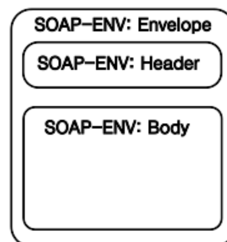


Figure 5.3: SOAP structure

Figure 5.3 illustrates the structure of SOAP. To simplify the specification, these four parts are functionally orthogonal. A SOAP message could be sent to a Web service (for example, a house price database) with the parameters needed for a search. The service would then return an XML-formatted document with the resulting data (prices, location, features, etc). Because the data is returned in a standardized machine-parsable format, it could then be integrated directly into a third-party site.

In particular, the envelope and the encoding rules are defined in different namespaces. Table 5.1 illustrates an example for a SOAP message.

```
<soapenv:Envelope
      xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <wsa:ReplyTo>
      <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:From>
      <wsa:Address>http://localhost:8080/axis2/services/MyService</wsa:Address>
    </wsa:From>
    <wsa:MessageID>ECE5B3F187F29D28BC11433905662036</wsa:MessageID>
  </soapenv:Header>
  <soapenv:Body>
    <req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">
      <req:category>classifieds</req:category>
    </req:echo>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 5.1: An XML fragment of response SOAP message

5.2.2 UDDI

Universal Description Discovery & Integration (UDDI) [23] is the definition of a set of services supporting the description and discovery of

- businesses, organizations, and other Web services providers,
- the Web services they make available,
- the technical interfaces, which may be used to access those services.

Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a Web services-based software environment for both publicly available services and services only exposed internally within an organization. 5.2 shows a piece of XML fragment of a Node Business Entity.

```
<businessEntity businessKey="uddi:tempuri.org:uddinodebusinessKey" xmlns="urn:uddi-org:api_v3">
   <name xml:lang="en">A UDDI Node</name>
   <description xml:lang="en"> This represents a sample model of how a UDDI node might represent itself in UDDI
   </description>
   <categoryBag>
        <keyedReference tModelKey="uddi:uddi.org:categorization:nodes" keyValue="node" />
   </categoryBag>
```

Listing 5.2: An XML fragment of UDDI entry

5.2.3 WSDL

WSDL is defined to describe the possible operations the basic Web service offered and the address or path to invoke it. WSDL is based on XML document, and specifies the Web Services in a machine-understandable way. WSDL defines *service* and *operation* to describe the interfaces, defines *type* and *messages* to describe the data types, and defines *port* and *binding* to describe the location of Web Services, etc[42]. Figure 5.4 illustrates the structure of a WSDL document.

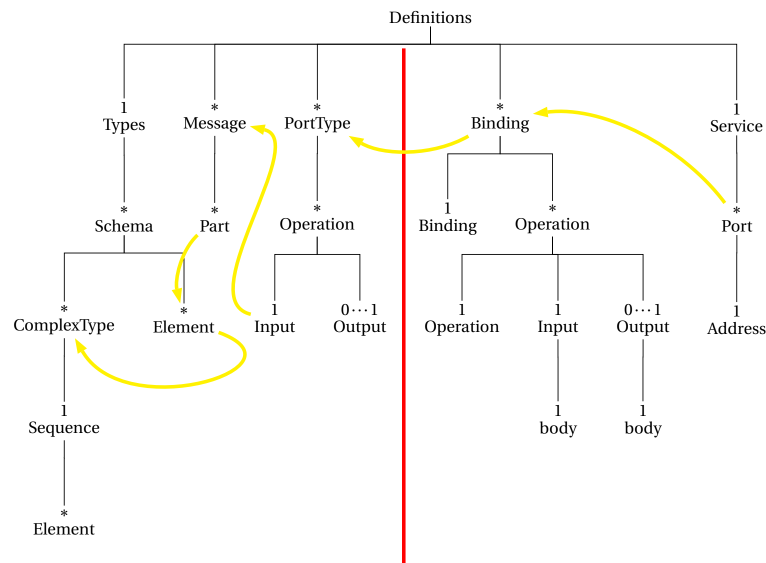


Figure 5.4: WSDL document structure: the yellow arrows represent the further definitions of the arrow sources, the red line separates the abstract definitions on the left from the concrete definitions on the right

```
1  <?xml version="1.0" encoding="UTF-8"?>
   <wsdl:definitions targetNamespace="http://localhost:8080/axis/services/delta" xmlns:wsdlsoap="..." xmlns:wsdl="...">
3    <wsdl:message name="calDeltaRequest">
       <wsdl:part name="in0" type="xsd:float"/>...
5    </wsdl:message>
     <wsdl:message name="calDeltaResponse">
7      <wsdl:part name="calDeltaReturn" type="xsd:float"/>
     </wsdl:message>
9    <wsdl:portType name="delta">
       <wsdl:operation name="calDelta" parameterOrder="in0 ...">
11       <wsdl:input name="calDeltaRequest" message="impl:calDeltaRequest"/>
         <wsdl:output name="calDeltaResponse" message="impl:calDeltaResponse"/>
13     </wsdl:operation>
     </wsdl:portType>
15   <wsdl:binding name="deltaSoapBinding" type="impl:delta">
       <wsdlsoap:binding style="rpc" transport="..." xmlns:wsdlsoap="..."/>
17     <wsdl:operation name="calDelta">
       <wsdlsoap:operation soapAction="" xmlns:wsdlsoap =...>
19       <wsdl:input name="calDeltaRequest">
```

```
      <wsdlsoap:body encodingStyle="..." namespace="http://converter" use="encoded"
21    xmlns:wsdlsoap="..."/>
        </wsdl:input>
23      <wsdl:output name="calDeltaResponse">
      <wsdlsoap:body encodingStyle="..." namespace="..." use="encoded" xmlns:wsdlsoap="..."/>
25      </wsdl:output>
      </wsdl:operation>
27  </wsdl:binding>
    <wsdl:service name="deltaService">
29    <wsdl:port name="delta" binding="impl:deltaSoapBinding">
      <wsdlsoap:address location="..." xmlns:wsdlsoap="..."/>
31    </wsdl:port>
    </wsdl:service>
33 </wsdl:definitions>
```

Listing 5.3: The code of WSDL of a basic Web service which calculate the value of delta of a quadratic equation

The listing 5.3 contains the WSDL file that describes a basic Web service that calculates the delta value of a second degree equation. It defines the name of the service as ”deltaService”, the service ”deltaService” offers an operation named ”calDelta” with the inputs ”in0”, ”in1”, ”in2”, and the output ”calDeltaReturn”. It also defines the ”binding” to describe the functioning mechanism of ”deltaService”. WSDL acts as a communication protocol of network services. It follows the XML grammar and also inherits extensible character of XML. So WSDL can be extended and enriched when necessary.

5.3 BPEL services

Business Process Execution Language (BPEL), short for Web Services Business Process Execution Language (WS-BPEL) is an OASIS standard executable language for specifying interactions with Web Services.

5.3.1 BPEL

A BPEL definition, like a structured programming language, consists of several parts describing partner links, process variables, correlation sets, the main process work-flow, the faults, and compensations handling activities.

The partner link declarations are used to define the relation between the process and its partners. It defines the role of the process in this relation (consumer or provider of an interface), and the interfaces used/provided by that role. The interfaces, operations, as well as their parameters and types, are specified in the corresponding WSDL documents.

The process variables are used to represent the state of the business process, they contain the information received from or sent to the partners of the process. The variables may be of primitive data types (e.g., strings, Boolean, integers) or of some complex types defined in a WSDL document.

The correlation sets define the parts of message data that are used to associate and route a particular message to a particular instance of the business process. Such information tokens uniquely identify the instance of the business process.

The process flow is defined by a set of process activities. They specify the operations to be performed, their ordering, conditional logic, reactive rules, etc. We distinguish the following groups of activities: basic activities, structured activities, and the specific operational blocks, namely faults and compensations handlers.

Basic activities represent primitive operations performed by the process, such as message emission/reception (invoke, receive, and reply activities), data modification (assign), process termination (terminate), waiting for a certain period of time (wait), or doing nothing (empty).

Structured activities define the order in which a collection of activities occurs. They compose the basic activities into structures that express the control flow patterns. The structured BPEL activities include sequence, switch, and while that model traditional control constructs; pick that models no deterministic choice based on external events (i.e., message reception or timeout); flow activity that models parallel execution of the nested activities. The structured activities can be recursively nested and combined.

Fault handling in BPEL is thought of as a mode switch from the normal processing. It is interpreted as *reverse work*, since it aims at undoing the unsuccessful work. The fault may arise on reception of the fault message, or on explicit invocation of the throw activity. The fault handler declaration specifies the activities to be performed when a fault arises.

The compensation handlers are used to reverse the effect of some unit of work that has completed with a fault. The compensation is always initiated within a fault handler, and may also require a compensation of some nested, previously successful, activities. A compensation handler is always associated with a work unit (BPEL scope), and is invoked (explicitly or implicitly) using the BPEL compensating activity.

5.3.2 Cooperation of BPEL and WSDL

BPEL introduces the features, e.g. process, action, correlation, role, partner link, etc, needed to describe the behavioral aspects of Web services. Figure 5.5 shows a sub-set of those features of interest in the context of this note and relationships between them.

A Web service may play multiple roles within a conversation. Usually, for each partner the Web service may expose a different role. BPEL partner link type defines binary relationship between roles. It specifies at most two roles that may communicate.

The BPEL is built on top of WSDL, which includes definitions of port types, messages and data types. Each role defined in the partner link type specifies exactly one WSDL port type it implements.

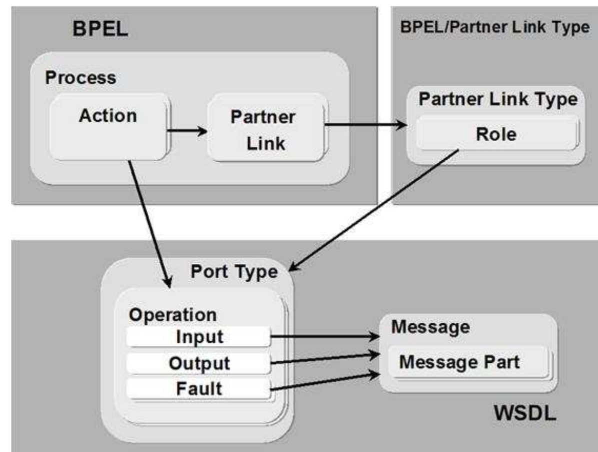


Figure 5.5: Interactivities between BPEL and WSDL protocols

5.3.3 ActiveBPEL engine

The ActiveBPEL engine [2] is an Open Source implementation of a BPEL engine, written in Java. It reads BPEL process definitions (and other inputs such as WSDL files) and creates graphical representations of each activities of BPEL processes. When an incoming message triggers a start activity, the engine creates a new process instance and starts running. The engine takes care of persistence, queues, alarms, and many other execution details. The ActiveBPEL engine runs in any standard servlet container such as Tomcat.

5.4 Case study: foodshop

The foodshop example is a foodshop company sells and delivers food. The company has an online SHOP (that does not have a physical counterpart) and several warehouses ($WH_1, \cdots, WH_n$, each of which is associated with their LocalSuppliers) located in different areas that are responsible for

stocking imperishable goods and physically delivering items to customers, depending on the area each customer lives in.

Customers ($C_1, \cdots, C_k$) interact with the foodshop company in order to place their orders, pay the bills and receive their goods.

In case of perishable items, that cannot be stocked, or in case of out-of-stock items, the foodshop company must interact with several suppliers ($SUP_1, \ldots, SUP_m$).

Although most of the interactions in this example are electronic, and take place between Web Services, in some cases there are physical actions and interactions that are performed by humans (e.g. the sending of a package). These too are modeled in the context of Web Services.

5.4.1 Partners interactions

In each conversation the following partners take part:

- one CUSTomer (represented in green);
- the online SHOP (represented in pink);
- one WAREHOUSE (represented in yellow);
- one LocalSupplier (represented in gray);
- a variable number of SUPPLIERS, which could also be 0 (represented in gray).

Figure 5.6 illustrates the BPEL processes of the partners of the foodshop example. When a CUSTomer places an order, the SHOP first selects the WAREHOUSE that is closest to the customer's address, and that will thus take part in the conversation. Ordered items are split into two categories: perishable (cannot be stocked, so the warehouse cannot possibly have them in stock) and imperishable (the warehouse might have them).

Perishable items are handled directly by the SHOP, while the WAREHOUSE handles the imperishable items. In case of the WAREHOUSE is out of stock, it can ask one of its LocalSupplier to fill the stock.

The first step is to check whether the ordered items are available, either in the warehouse or from the suppliers (we have not considered items exchanges among different warehouses, in order not to make the example too complicated). If they are, they are temporarily reserved in order to avoid conflicts between several orders.

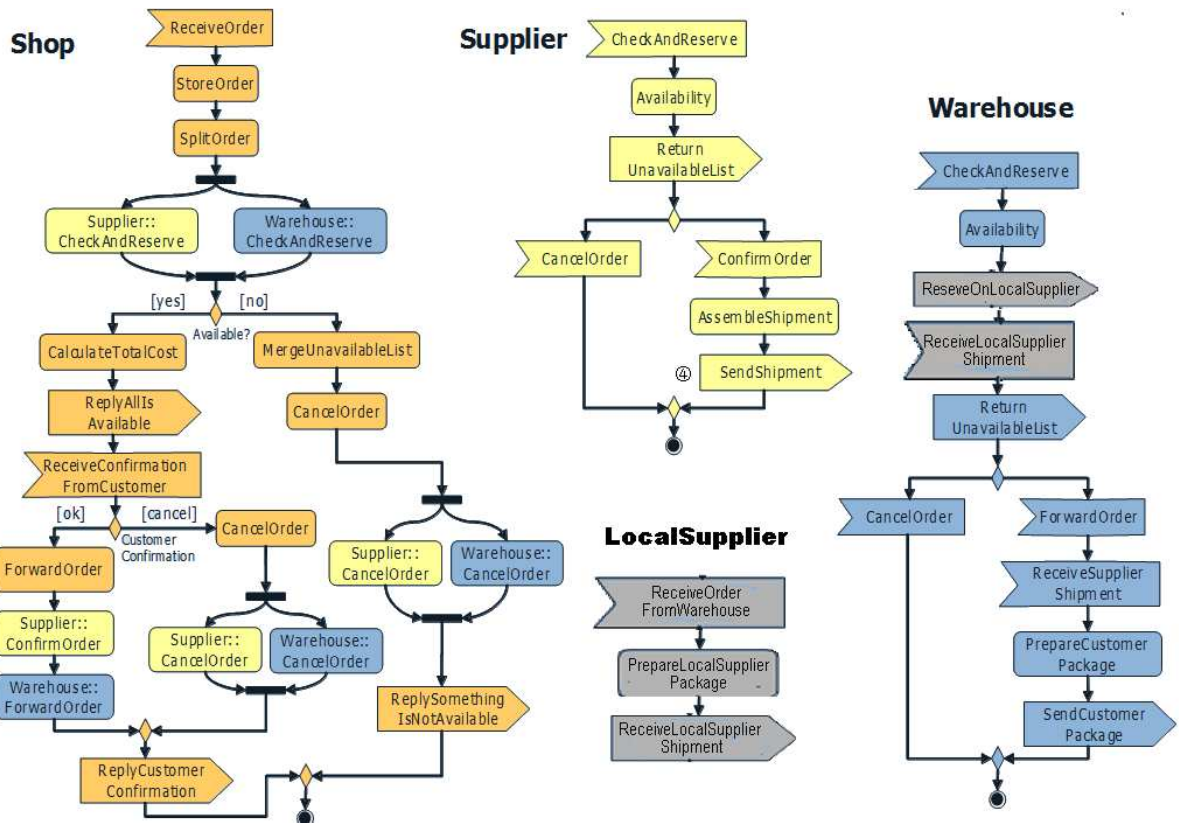


Figure 5.6: The partners of the foodshop example: the graphical representations run over the ActiveBPEL engine is much more specific and in detail.

Once the SHOP receives all the answers on availability, it can decide whether to give up with the order (again, in order to keep things simple, this happens whenever there is at least one unavailable item) or to proceed. In the former case, all item reservations are canceled and the conversation ends.

If the order goes on, the SHOP computes the total cost (items + shipping) with the aid of the WAREHOUSE that provides the shipping costs. Then it sends the bill to the CUSTomer, that can decide whether to pay or not. If the CUSTomer does not pay, all item reservations are canceled and the conversation ends here.

If the CUSTomer pays, then all item reservations are confirmed and all the SUPPLIERS (in case of perishable or out-of-stock items) are asked to send the goods to the WAREHOUSE. The WAREHOUSE will then assemble a package and send it to the CUSTomer.

5.4.2 BPEL services execution processes

We describe separately the execution processes of each partner, including its interactions with others in the same composite process.

Customer

The customer workflow (figure 5.7) is abstract: we represent only its interface with the other services, while we do not represent internal activities. The CUSTomer places an order ($sendOrder$) communicating the items he/she is interested in ($items$) and its personal data ($custInfo$). Then it waits for an answer from the SHOP: if some of the items are not available the conversation ends ($exit$). Otherwise the user receives the bill and decides whether to pay ($replyPay$) sending its payment to the SHOP.

If the CUSTomer decides not to pay the conversation ends ($exit$). Otherwise, he/she waits for the parcel sent by one of the company's WAREHOUSEs. Notice that the parcel shipment is a physical transaction, while the others are all electronic transactions.

Shop service

On the contrary, the SHOP workflow (in figure 5.6) is detailed, and contains several internal activities.

When the SHOP receives an order ($receiveOrder$) with the ordered items and the CUSTomer data ($custInfo$), it selects the WAREHOUSE that is closer to the user ($selectWH$) and splits ($splitOrder$) the ordered items into the set of perishable items (ns_items) and that of imperishable

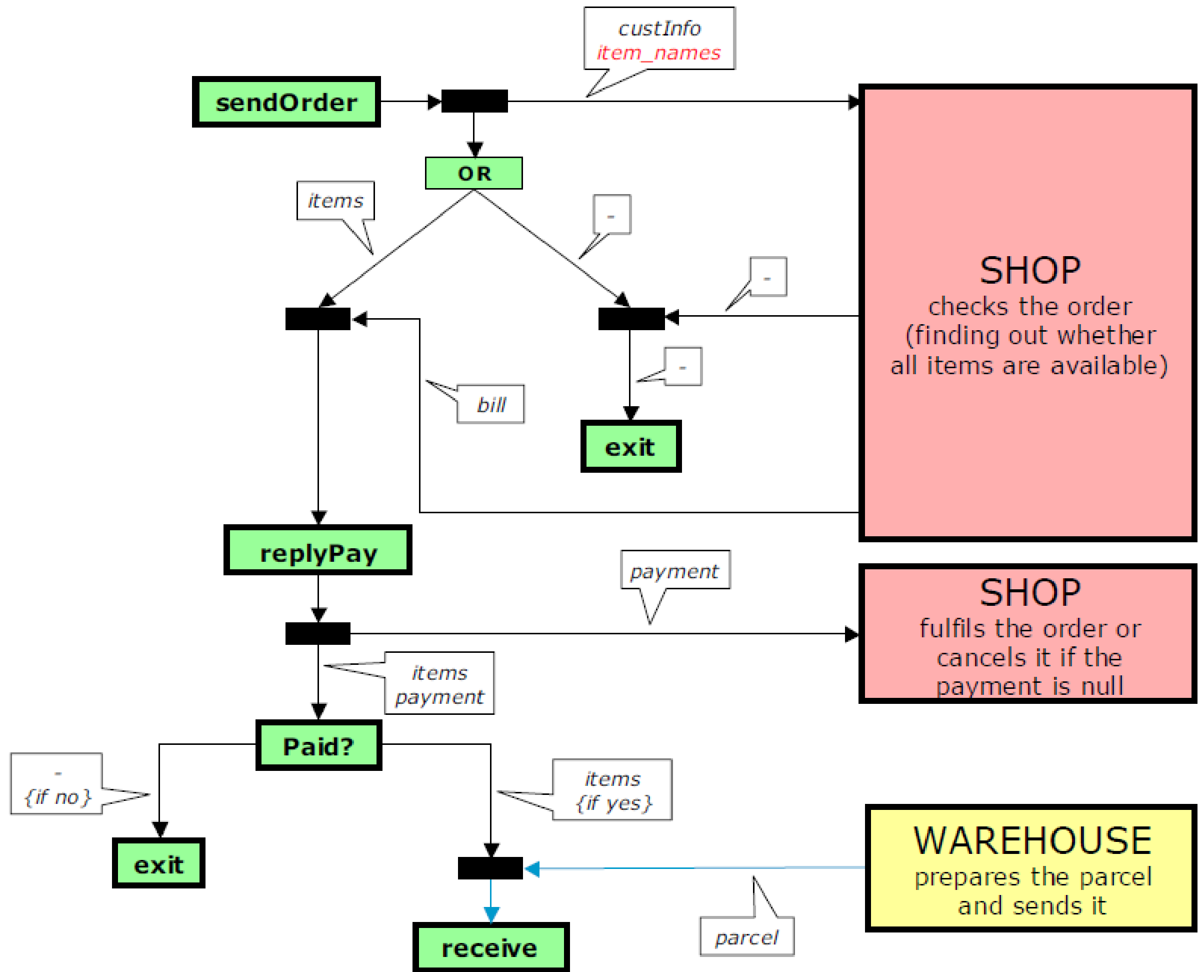


Figure 5.7: The workflow of customer

items (s_items). It then checks the availability of perishable items ($checkAvail\&reserve$) with the SUPPLIERS, asking to temporarily reserve them in case they are available. The SHOP receives back the set of reserved items ($ns_resitems$), the corresponding reservation codes ($ns_answers$), and the answers on availability ($ns_answers$).

The list of imperishable items is instead sent to the WAREHOUSE ($checkAvail$), that sends back a collective answer ($s_answers$) on availability.

If any of the items is unavailable, the order is canceled. The SHOP communicates this to the CUSTomer, and cancels the reservations ($unreserved$) both with the SUPPLIERS and the WAREHOUSE.

If on the other hand all the items are available, the SHOP asks the WAREHOUSE to compute the ship cost ($shipCost$), which depends on the distance between the WAREHOUSE itself and the user address, as well as the total weight of the ordered items (for this reason, the SHOP sends to the WAREHOUSE both the list of $items$ and $custInfo$).

Then the SHOP computes the total cost ($totalCost$) and sends the bill to the CUSTomer, which sends back a payment. if the CUSTomer decides not to pay, the SHOP cancels all the reservations ($unreserved$) with the SUPPLIERS and the WAREHOUSE. If the payment is OK, the SHOP forwards the order to the WAREHOUSE ($fwOrder$), which from now on is responsible for it, and tells the SUPPLIERS to send the reserved items to the WAREHOUSE ($requestSupply$), providing the reservation codes ($ns_rescodes$) and the warehouse address ($whInfo$).

Realsupplier service

Like the CUSTomer workflow, the SUPPLIER workflow (in figure 5.6) is abstract since each supplier may have a different internal workflow.

Of course, it is the same workflow independently from the Web Service that contacts the SUPPLIER. For this reason, the Web Service that buys the goods is generically called BUYER, while the receiver of the products is generically called RECEIVER. It is clear that in our context the BUYER can be either the SHOP or the WAREHOUSE, while the RECEIVER is always the WAREHOUSE.

The SUPPLIER is first asked by the user to verify the availability of some items and reserve them ($verify\&reserve$). The SUPPLIER sends back the set of reserved items ($resitems$), the corresponding reservation codes ($rescodes$) and the answers on availability.

Then the BUYER can either cancel the reservation ($unReserve$) or ask the SUPPLIER to send the items ($supply$) to the address ($sendAddress$) of the RECEIVER.

Warehouse service and LocalSupplier service

The WAREHOUSE first receives a request from the SHOP to check the availability of some items (s_items) and reserve them ($reserveAvail$). If some items are out-of-stock, the WAREHOUSE contacts the LocalSuppliers in order to check for availability and to reserve them ($findSuppliers$), receiving back the set of reserved items ($s_resitems$), the corresponding reservation codes ($s_rescodes$) and the answers on availability ($s_answers$).

The WAREHOUSE elaborates a collective answer on availability and sends it to the SHOP ($collectAnswers$). Then it waits for one of the following things to happen: either the SHOP decides to cancel the order, or to proceed.

In the first case the WAREHOUSE has to cancel its own reservations, and, in case some LocalSuppliers were contacted, it must also cancel the reservations with the LocalSupplier ($unreserved$).

In the second case, the WAREHOUSE is asked by the SHOP to compute the shipment cost. Then the SHOP tells the WAREHOUSE to proceed with the order. In case of out-of-stock items, the WAREHOUSE asks the SUPPLIER to send the reserved items ($requestSupply$), by providing the reservation codes ($s_rescodes$) and its address ($whInfo$).

At this point the WAREHOUSE must assemble the package. In order to do this, it must wait both for the ($unperishable$) items it reserved directly from the SUPPLIER, and for the ($perishable$) items that were reserved by the SHOP from the SUPPLIER.

Once the parcel is ready, the WAREHOUSE asks a shipper ($requestShipping$) to send it to the user.

5.5 Translate from BPEL to CPN

A BPEL process consists of basic activities and structured operators. The idea of modeling BPEL to CPN is: to map each primitive data to a place, each basic activity to a transition (except that a two-way $Invoke$ activity is mapped to two transitions to represent the interaction between the partners). To each basic activity, local input and output activation place $a^{in} \in P$ and $a^{out} \in P$ are associated to identify the local execution order; a remote input and/or output activation places $a^{in}_{PA} \in P$ and/or $a^{out}_{PA} \in P$ is added to allow the remote control, and a set of remotely shared data places $P^r \subseteq P$ (messages) are marked also. The fault (KO) mode(s) of transitions are added on the T/P output arc expression to represent the unobservable faulty activities either in basic Web services or in BPEL services. The structured operators to sew up the structured sub-processes by combining, disjointing, or generating the local activation places. Once a red token is generated by

the KO mode of a transition of a basic activity, the fault is passed along the execution trace through the data dependency functions. Figure 5.8 illustrates the basic places and transitions we used for translation from BPEL to CPN.

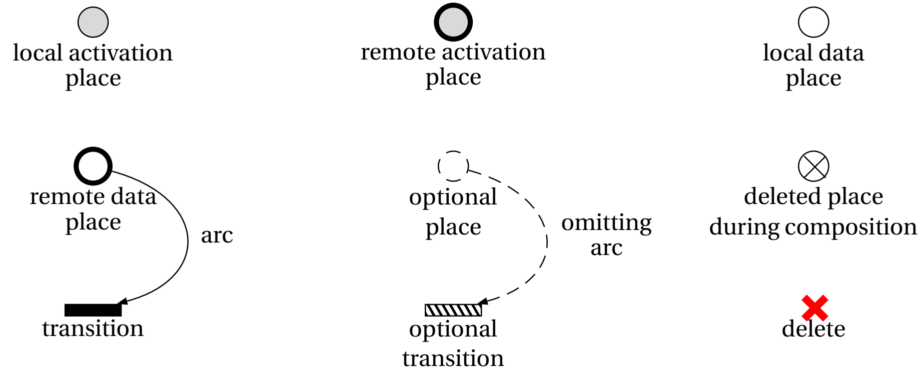


Figure 5.8: basic place/transition representations

We assume the BPEL services studied in this thesis are basically debugged and ready to run, so assumption 3 makes sure the CPN diagnosis is based on the well-formed CPN model:

Assumption 3. *The BPEL processes are realized according to their specifications and basically debugged* .

In the following sections, we model dynamic features, the basic BPEL activities and structured operators, with CPNs.

5.5.1 Translating static BPEL features to CPNs

BPEL processes manipulate a set of data, variables, and constants. Variables are typed either by primitive types (int, float, string, etc.), or by structured ones (complex type structure of XML), which are normally defined in the associated WSDL files.

To catch maximally the dependency between data (variables, constants, etc.), we decompose the structured data types into their elementary parts, denoted by the $leaves$ of their XML tree structure. For a variable X of type m (resp. an Xpath expression), we use x_i to range over the $Leaves(m)$ (resp. $Leaves(X)$) and denote the x_i part of X by a couple (X, x_i). In our mapping, each data variable and constant is represented by a unique place in CPNs.

5.5.2 Translation from basic Web service to CPN

Basic Web service

A basic Web service is a program, which publishes its invocation interface and can be remotely called by other Web service, including a BPEL service. As it is called synchronously through its WSDL interface and cannot be decomposed, we model it as a CPN system, which has a transition, remote input/output activation places and a set of shared input/output data places (all the local components are in the dotted line boxes). The data dependency between its input and output can be FW, EL, and/or SRC, which should be offered by the basic Web service developers. The CPN model of a basic Web service (see figure 5.9) is a transition t_B, which can be in OK or KO mode. They are triggered by the consummation of the token in the output activation place. Once the transition is enabled and behaves in KO mode, there should be a fault in its output data place and the fault can be passed to its invoker, a BPEL process, when it receives the response of the basic Web service. So the formal definition of CPN fault model for a basic Web service is as follows:

Definition 59 (CPN fault model for basic WS)**.** *A CPN fault model for basic Web service is a tuple* N_B=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩*, where*[1]*:*

- $P = \{a^{in}_PA, a^{out}_PA, x_i_PA, y_i_PA | i \in I\}$ *is a set of labeled places of type* $\{b, r, *\}$[2]*;*
- $T = \{t_B\}$*;*
- $md(t_B) = \{OK, KO\}$*;*
- Pre *and* $Post$ *are illustrated in table 5.1;*

Receive(m, X)

Receive is an activity simply copies the values from a message m sent from a partner service to a local variable X. So the data dependency relation is defined ad FW and there is no KO mode for the transition. We model $Receive$ as the CPN presented in figure 5.11. Data places $(m, m_i), (x, x_i)$ are simplified as m_i, which represents the shared message with its partner PA_i; and x_i, which represents the variable, which saves the value of m_i. A remote activation place a^{in}_{PA} is added to

[1]Without especially defined, the definition is same as the definition of the CPN fault model 33 in chapter 3

[2]In the following sections, not specially claimed, denote a^{in} and a^{out} as the input and output activation places, x_i as a local input data place, y_i as a local output data place, m_i as a remote input or output date place. $_PA$ is use to indicate the remote place.

Pre	t_B	
	OK	KO
a^{in}_PA	$\chi_{a^{in}_PA}$	$\chi_{a^{in}_PA}$
a^{out}_PA		
x_i	χ_{x_i}	χ_{x_i}
y_i	χ_{y_i}	χ_{y_i}

$Post$	t_B	
	OK	KO
a^{in}_PA		
a^{out}_PA	$FW(\chi_{a^{out}_PA})$	$FW(\chi_{a^{out}_PA})$
x_i	χ_{x_i}	χ_{x_i}
y_i	$F(\chi_{x_i})$	r

Table 5.1: Pre and $Post$ tables of a basic Web service: $F(\chi_{x_i})$ represents a data dependency function, which can be FW, SRC, or EL. So for SRC, there is no input.

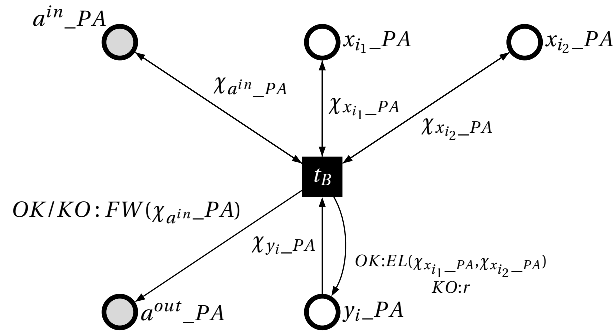


Figure 5.9: CPNs of the basic activity: the thick-line places represent the remote places

activate the $Receive$ activity. The color of the output activation place a^{out} is decided by the remote activation input place, which means the wrong work flow is chosen. So the arc expression on the arc (t_{rec}, a^{out}) is $FW(\chi_{a^{in}_PA})$. To keep the liveness of the CPN, we add an arc from the output place x_i to the receive transition t_{rec} and its associated color function χ_{x_i} is simply the color of the output data place x_i.

So the formal definition of CPN fault model for a $Receive$ activity is as follows:

Definition 60 (CPN fault model for $Receive$ activity)**.** *A CPN fault model for Receive activity is a tuple* N_{rec}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩, *where:*

- $\Gamma = \{OK\}$;
- $P = \{a^{in}_PA, a^{out}_PA, a^{in}, a^{out}, m_i_PA, x_i | i \in I\}$;
- $T = \{t_{rec}\}$;
- $md(t_{rec}) = \{OK\}$;
- *Pre and $Post$ are illustrated in table 5.10;*

Pre	t_{rec}
	OK
a^{in}_PA	$\chi_{a^{in}_PA}$
a^{in}	$\chi_{a^{in}}$
a^{out}	
m_i	χ_{m_i}
x_i	χ_{x_i}

$Post$	t_{rec}
	OK
a^{in}_PA	
a^{in}	
a^{out}	$FW(\chi_{a^{in}_PA})$
m_i	χ_{m_i}
x_i	$FW(\chi_{m_i})$

Figure 5.10: Pre and $Post$ tables of a $Receive$ activity

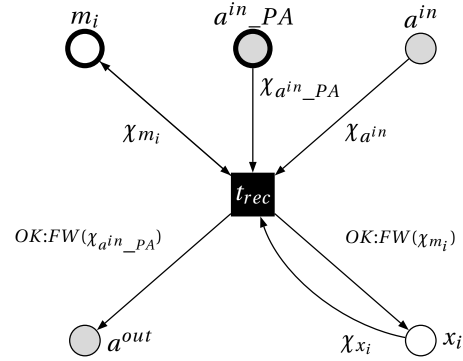


Figure 5.11: CPNs of the receive activity: the thick-line places represent the remote places

Invoke(X, Y)

Invoke is an activity that calls another Web service, either a basic or a composite one. It takes the value of the variable X, sends a remote request to its partner, synchronously or asynchronously waits for the response message, stores it in the variable Y, or gets the response by $Receive(m, Y)$. Y can be infected by external faulty Web service (basic or composed), which is locally unobservable. a KO mode of transition t_{inv} to model the faults caused by the external Web service. $Invoke$ can be a one-way call, which only sends a request X to another Web service. In this case, the data dependency between the input and output data is FW and the transition only has OK mode. When $Invoke$ is a two-way call, it has both the local and remote input/output activation and data places. When it invokes a composed Web service, the data dependencies between the remote places depend on the invoked Web service. So the data dependencies between the local and remote input places are FW, such as between the local and remote output places. When it invokes a basic Web service, the data dependency can be retrieved directly from the published WSDL interface of the basic Web service. But the synchronous $Invoke$ activity has both the OK and KO modes. The figure 5.12b and 5.12a respectively illustrate the CPN model of a one-way and two-way $Invoke$ activity.

So the formal definition of CPN fault model for an asynchronous $Invoke$ activity is as follows:

Definition 61 (CPN fault model for an asynchronous $Invoke$ activity)**.** *A CPN fault model for an asynchronous $Invoke$ activity is a tuple N_{asInv}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩, where:*

- $P = \{a^{out}_PA, a^{in}, a^{out}, x_i, m_i^{out}_PA | i \in I\}$;
- $T = \{t_{inv}\}$;
- *Pre and $Post$ are illustrated in table 5.2;*

Pre	t_{inv}
	OK
a^{in}	$\chi_{a^{in}}$
a^{out}	
a^{out}_PA	
x_i	χ_{x_i}
m_i^{out}	$\chi_{m_i^{out}}$

$Post$	t_{inv}
	OK
a^{in}	
a^{out}	$FW(\chi_{a^{in}})$
a^{out}_PA	$FW(\chi_{a^{in}})$
x_i	χ_{x_i}
m_i^{out}	χ_{x_i}

Table 5.2: Pre and $Post$ tables of an asynchronous $Invoke$ activity

The formal definition of CPN fault model for a synchronous $Invoke$ activity is as follows:

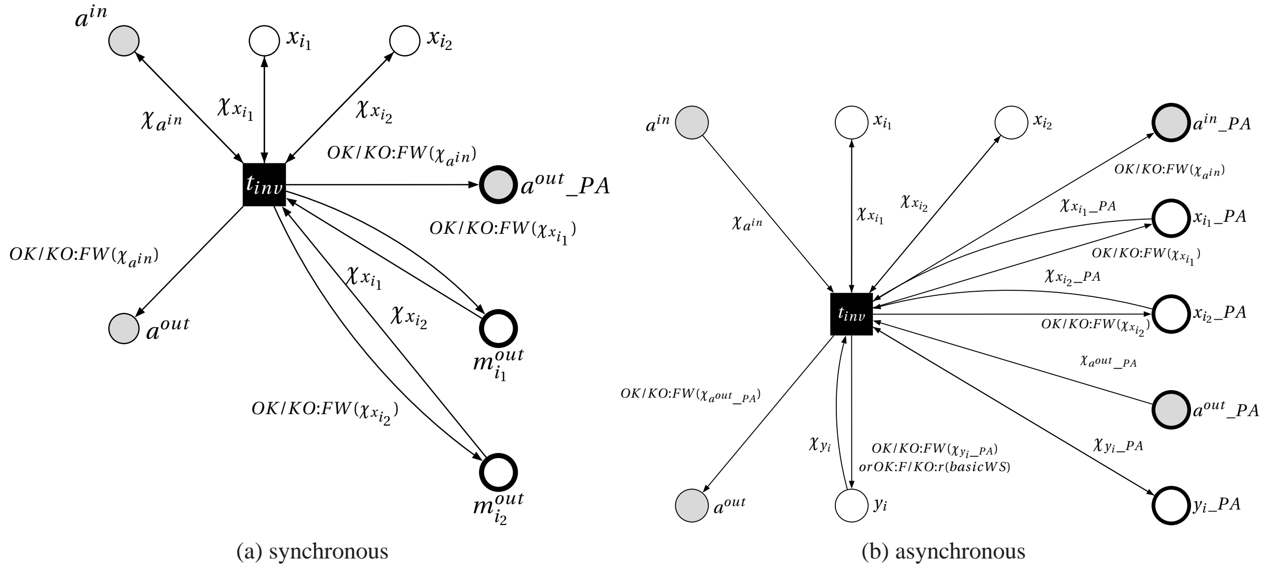
Definition 62 (CPN fault model for a synchronous $Invoke$ activity)**.** *A CPN fault model for a synchronous $Invoke$ activity is a tuple N_{inv}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩, where:*

- $P = \{a^{in}_PA, a^{out}_PA, a^{in}, a^{out}, x_i, y_i, x_i_PA, y_i_PA | i \in I\}$;
- $T = \{t_{inv}\}$;
- *Pre and $Post$ are illustrated in table 5.3;*

Pre	t_{inv}	
	OK	KO
a^{in}	$\chi_{a^{in}}$	$\chi_{a^{in}}$
a^{in}_PA		
a^{out}_PA	$\chi_{a^{out}_PA}$	$\chi_{a^{out}_PA}$
a^{out}		
x_i	χ_{x_i}	χ_{x_i}
x_i_PA	$\chi_{x_i_PA}$	$\chi_{x_i_PA}$
y_i	χ_{y_i}	χ_{y_i}
	χ_{y_i}	χ_{y_i}
y_i_PA	$\chi_{y_i_PA}$	$\chi_{y_i_PA}$

$Post$	t_{inv}	
	OK	KO
a^{in}		
a^{in}_PA	$FW(\chi_{a^{in}})$	$FW(\chi_{a^{in}})$
a^{out}_PA		
a^{out}	$FW(\chi_{a^{out}_PA})$	$FW(\chi_{a^{out}_PA})$
x_i	χ_{x_i}	χ_{x_i}
x_i_PA	$FW(\chi_{x_i})$	$FW(\chi_{x_i})$
y_i	$FW(\chi_{y_i_PA})$	$FW(\chi_{y_i_PA})$
	F	r
y_i_PA	$\chi_{y_i_PA}$	$\chi_{y_i_PA}$

Table 5.3: Pre and $Post$ tables of a synchronous $Invoke$ activity: F denotes the data dependency function which depends on the invoked basic Web service



(a) synchronous (b) asynchronous

Figure 5.12: CPN model of invoke activity: the thick-line places represent the remote places

Reply(Y, m)

Reply is an activity that copies values from a variable Y to a message m for returning the response of the BPEL service to its invoker. So $Reply$ simply forwards (FW) values (figure 5.14) for the OK mode and there is no KO mode. The data dependency function on the arc from t_{rep} to the remote activation place a^{out}_{PA} is $FW(\chi_{a^{in}_{PA}})$, because the remote activation does not affect the activation correctness of the local process. The formal definition of CPN fault model for a $Reply$ activity is as follows:

Definition 63 (CPN fault model for $Reply$ activity)**.** *A CPN fault model for $Reply$ activity is a tuple N_{rep}=$\langle\Sigma$, Γ, P, T, Pre, $Post$, F,$\rangle$, where:*

- $\Gamma = \{OK\}$;
- $P = \{a^{out}_PA, a^{in}, a^{out}, y_i, m_i_PA | i \in I\}$;
- $T = \{t_{rep}\}$;
- $md(t_{rep}) = \{OK\}$;
- *Pre and $Post$ are illustrated in table 5.13;*

Pre	t_{rep}
	OK
a^{out}_PA	
a^{in}	$\chi_{a^{in}}$
a^{out}	
y_i	χ_{y_i}
m_i	χ_{m_i}

$Post$	t_{rep}
	OK
a^{out}_PA	$FW(\chi_{a^{in}})$
a^{in}	
a^{out}	$FW(\chi_{a^{in}_PA})$
y_i	χ_{y_i}
m_i	$FW(\chi_{y_i})$

Figure 5.13: Pre and $Post$ tables of a $Reply$ activity

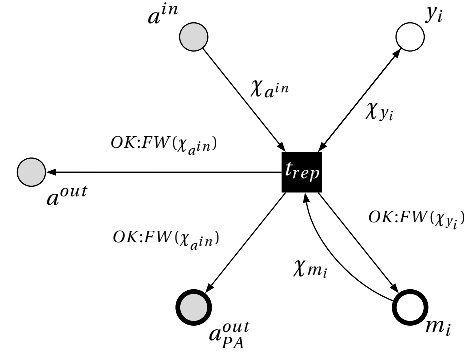


Figure 5.14: CPNs of the reply activity: the thick-line places represent the remote places

Expression(C, V)

Expression is an internal activity to calculate a variable x_i based on some constants c_j and other variables v_k. Expression is usually defined in assign activity and other structural operators which to define the sub process choice conditions, like switch, while, if, etc. According to the assumption 3, transition t_{exp} behave always in OK mode. The formal definition of CPN fault model (see figure 5.16) for a basic Web service is as follows:

Definition 64 (CPN fault model for expression)**.** *A CPN fault model for basic Web service is a tuple* N_{exp}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩*, where:*

- $P = \{a^{in}, a^{out}, c_j, v_k, x_i | i, j, k \in I\}$;
- $T = \{t_{exp}\}$;
- $md(t_{exp}) = \{OK, KO\}$;
- Pre *and* $Post$ *are illustrated in table 5.15;*

Assign(X, Y)

Assign is an activity that reorganizes the variable parts to compose the new ones. So its model does not contain KO mode, remote activation, or shared data places. And the data dependency between the input and output places is FW. The formal definition of CPN fault model (see figure 5.17) for $Assign$ is as follows:

Definition 65 (CPN fault model for $Assign$)**.** *A CPN fault model for* $Assign$ *is a tuple* N_{exp}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩*, where:*

Pre	t_{exp}	
	OK	KO
a^{in}	$\chi_{a^{in}}$	$\chi_{a^{in}}$
a^{out}		
c_j	χ_{c_j}	χ_{c_j}
v_k	χ_{v_k}	χ_{v_k}
x_i	χ_{x_i}	χ_{x_i}

$Post$	t_{exp}	
	OK	KO
a^{in}		
a^{out}	$FW(\chi_{a^{out}})$	$FW(\chi_{a^{out}})$
c_j	χ_{c_j}	χ_{c_j}
v_k	χ_{v_k}	χ_{v_k}
x_i	F	r

Figure 5.15: Pre and $Post$ tables of $Expression$ activity: F represents a data dependency function, which can be either FW, SRC, or EL with the input(s) as c_j and/or v_k. For SRC, there is no input.

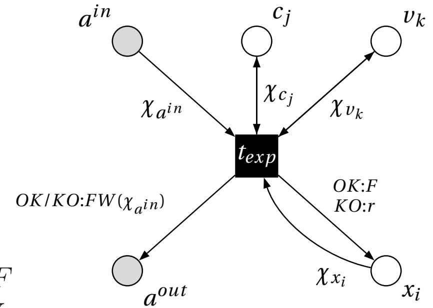


Figure 5.16: CPNs of the expression activity

- $P = \{a^{in}, a^{out}, x_i, y_i | i \in I\}$;
- $T = \{t_{ass}\}$;
- $md(t_{ass}) = \{OK\}$;
- Pre *and* $Post$ *are illustrated in table 5.4;*

Pre	t_{ass}
	OK
a^{in}	$\chi_{a^{in}}$
a^{out}	
x_i	χ_{x_i}
y_i	χ_{y_i}

$Post$	t_{ass}
	OK
a^{in}	
a^{out}	$FW(\chi_{a^{in}})$
x_i	χ_{x_i}
y_i	$FW(\chi_{x_i})$

Table 5.4: Pre and $Post$ tables of $Assign$ activity

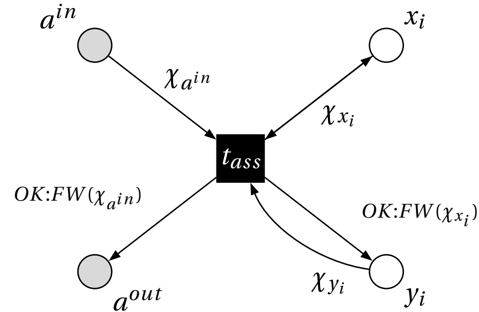


Figure 5.17: CPNs of the assign activity

In fact, $Expressions$ are always defined in $Assign$ to generate the temporal variables as the input of the $Assign$ activity. So the activities $Expression$ and $Assign$ can be united

Throw/Rethrow$(faultName, [faultVariable])$

$Throw$ and $rethrow$ are activities to signal an internal fault explicitly. They provide the name for the fault, and can optionally provide data with further information about the fault. So their input places can be the "faultName", and the optional "faultVariable" and no output data places are generated. The figure 5.18 illustrates the CPN model of a $throw$ or a $rethrow$ activity. Note that $rethrow$ ignores the modifications to the fault data and throws the original message type data. $Rethrow$ can be used only within a fault handler ($catch$ and $catchAll$ in section 5.5.4). The formal

definition of CPN fault model (see figure 5.18) for $Throw$ (or $Rethrow$) is as follows:

Definition 66 (CPN fault model for $Throw$)**.** *A CPN fault model for $Throw$ is a tuple N_{thr}=⟨Σ, Γ, P, T, Pre, Post, F,⟩, where:*

- $P = \{a^{in}, a^{out}, faultName, faultVariable\}$;
- $T = \{t_{thr}\}$;
- $md(t_{thr}) = \{OK\}$;
- *Pre and $Post$ are illustrated in table 5.5;*

Pre	t_{thr}
	OK
a^{in}	$\chi_{a^{in}}$
a^{out}	
$faultName$	$\chi_{faultName}$
$faultVariable$	$\chi_{faultVariable}$

$Post$	t_{thr}
	OK
a^{in}	
a^{out}	$FW(\chi_{a^{in}})$
$faultName$	$\chi_{faultName}$
$faultVariable$	$\chi_{faultVariable}$

Table 5.5: Pre and $Post$ tables of $Throw$ activity

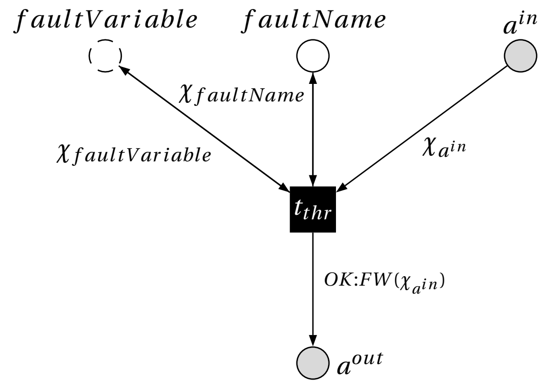


Figure 5.18: CPNs of the throw/Rethrow activity: the dashed-line places represent the optional places

Wait$(duration|until)$

The wait activity specifies a delay for a certain period of time or until a certain deadline is reached. The time delay is modeled as a input data place, no input or output data place is defined. The figure 5.19 illustrates the CPN model of a $Wait$ activity. The formal definition of CPN fault model (see figure 5.19) for $Wait$ is as follows:

Definition 67 (CPN fault model for $Wait$)**.** *A CPN fault model for $Wait$ is a tuple N_{wait}=⟨Σ, Γ, P, T, Pre, Post, F,⟩, where:*

- $P = \{a^{in}, a^{out}, duration|until\}$;
- $T = \{t_{wait}\}$;
- $md(t_{wait}) = \{OK\}$;
- *Pre and $Post$ are illustrated in table 5.6;*

Pre	t_{wait}
	OK
a^{in}	$\chi_{a^{in}}$
a^{out}	
$faultName$	$\chi_{duration\|until}$

$Post$	t_{wait}
	OK
a^{in}	
a^{out}	$FW(\chi_{a^{in}})$
$faultName$	$\chi_{duration\|until}$

Table 5.6: Pre and $Post$ tables of $Wait$ activity

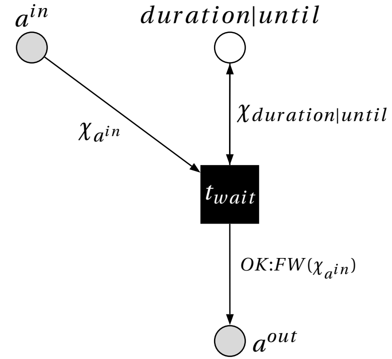


Figure 5.19: CPNs of the wait activity: the dashed-line places represent the optional places

Pre	t_{emp}
	OK
a^{in}	$\chi_{a^{in}}$
a^{out}	

$Post$	t_{emp}
	OK
a^{in}	
a^{out}	$FW(\chi_{a^{in}})$

Table 5.7: Pre and $Post$ tables of $Throw$ activity

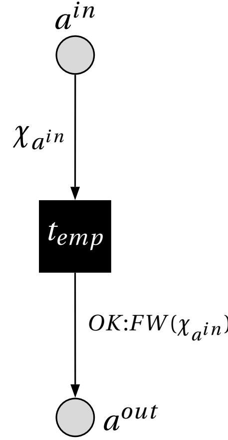


Figure 5.20: CPNs of the $Empty$ activity: the dashed-line places represent the optional places

Empty

Definition 68 (CPN fault model for $Empty$)**.** *The $Empty$ activity does nothing, for example when a fault needs to be caught and suppressed, or to provides a synchronization point in a synchronous subprocess. So the CPN model of $Empty$ activity has no data place. The figure 5.20 illustrates the CPN model of an $Empty$ activity. The formal definition of CPN fault model (see figure 5.20) for $Empty$ is as follows:*

A CPN fault model for $Empty$ is a tuple N_{emp}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩, where:

- $P = \{a^{in}, a^{out}\}$;
- $T = \{t_{emp}\}$;
- $md(t_{emp}) = \{OK\}$;
- *Pre and $Post$ are illustrated in table 5.7;*

Exit

The $Exit$ activity immediately ends the business process instance, includes the synchronous siblings. All currently running activities MUST be ended immediately without involving any termination handling, fault handling, or compensation behavior. So the CPN model of $Exit$ has no output activation place but may have the extra input activation places, which are the output activation places of their synchronous siblings processes. The figure 5.21 illustrates the CPN model of an $Exit$ activity. The formal definition of CPN fault model (see figure 5.21) for $Exit$ is as follows:

Definition 69 (CPN fault model for $Exit$)**.** *A CPN fault model for* $Exit$ *is a tuple* N_{exit}=⟨Σ, Γ, P, T, Pre, $Post$, F,⟩, *where:*

- $P = \{a^{in}, a^{out}_{sib}\}$;
- $T = \{t_{exit}\}$;
- $md(t_{exit}) = \{OK\}$;
- Pre *and* $Post$ *are illustrated in table 5.5;*

Pre	t_{exit}
	OK
a^{in}	$\chi_{a^{in}}$
a^{out}_{sib}	$\chi_{a^{out}_{sib}}$

$Post$	t_{exit}
	OK
a^{in}	
a^{out}_{sib}	

Table 5.8: Pre and $Post$ tables of $Exit$ activity

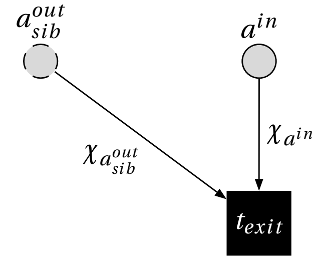


Figure 5.21: CPNs of the $Exit$ activity: the dashed-line places represent the optional places

5.5.3 Structured operators translation

Sequence operator $sequence(N_1, N_2)$

$Sequence$ connects different activities, and the execution order of these activities is the same as their appearance order in the constructor. Given two sub-processes N_1 N_2 in a sequence structure, with N_1=⟨Σ_1, Γ_1, P_1, T_1, Pre_1, $Post_1$, F_1 ⟩, and N_2=⟨Σ_2, Γ_2, P_2, T_2, Pre_2, $Post_2$, F_2 ⟩, So we can generate the resulting sequence CPN by simply merging the local intermediate output and input activation places of contractive CPNs (in figure 5.22.

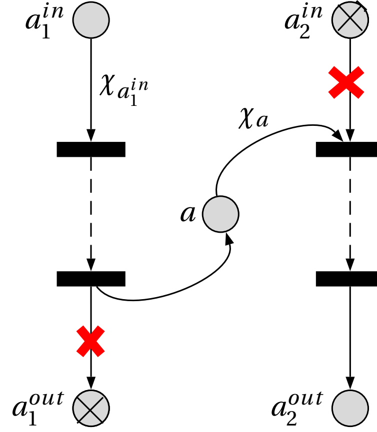


Figure 5.22: CPNs of the sequence activity: the red crosses represent the canceled arc

Conditional operator $Switch(\{(con_i(X_i, V_i),N_i)\}_{i\in I})$

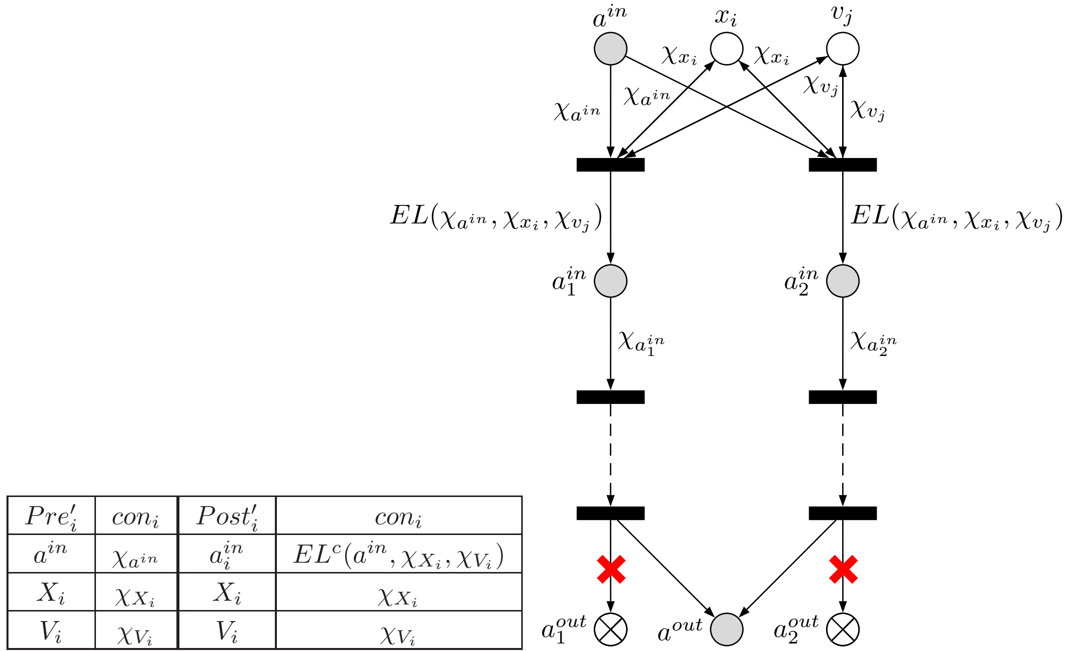
$Switch$ represents an alternative execution of the activities N_i under the conditions $con_i(X_i, V_i)$. X_i and V_i are respectively the variables and constants. For each sub process N_i, we add a transition con_i to generate its activation place. Each con_i takes the common activation input place of $Switch$, X_i, and V_i as inputs to elaborate an input activation place a_i^{in} for sub process N_i. Let N_i=$\langle\Sigma_i$, Γ_i, P_i, T_i, Pre_i, $Post_i$, F_i $\rangle$. We define the additional tables Pre'_i for Pre_i and $Post'_i$ for $Post_i$ in tables in 5.23. The CPN graph of the resulting activity is N=$\langle\Sigma$, Γ, P, T, Pre, $Post$, F $\rangle$ with: Σ=$\bigcup_{i\in I}\Sigma_i$, Γ=$\bigcup_{i\in I}\Gamma_i$, P=$\bigcup_{i\in I}(P_i\backslash\{a_i^{out}\}\cup$ $X_i\cup V_i)\cup$ $\{a^{in}$,$a^{out}\}$, T=$\bigcup_{i\in I}(T_i\cup\{t_{con_i}\})$, F=$\bigcup_{i\in I}F_i$, , Pre=$\bigcup_{i\in I}(Pre_i\oplus Pre'_i)$, $Post$=$\bigcup_{i\in I}(Post_i\oplus Post'_i)$[3] (in figure 5.24.

Iterative operator $while(con(X, V), S_1)$

$While$ iterates the activity S_1 execution until the breaking off of the conditions $con(X, V)$. The CPN graph of $While$ is similar to $Switch$ in which the activation input place of the sub process S_1 is elaborated by the activation input place of $While$, X, and V. But in $While$, the a^{out} of the iterative sub process is also a^{in} of t_{con}.

Note that t_{con} represents the transition if condition con is true and $t_{\overline{con}}$ represents the transition if condition con is false (in figure 5.25a).

[3] $C\oplus C'$ means to update C with C' by adding places' rows and transitions' columns)



Pre'_i	con_i	$Post'_i$	con_i
a^{in}	$\chi_{a^{in}}$	a_i^{in}	$EL^c(a^{in}, \chi_{X_i}, \chi_{V_i})$
X_i	χ_{X_i}	X_i	χ_{X_i}
V_i	χ_{V_i}	V_i	χ_{V_i}

Figure 5.23: Additional Pre_i and $Post_i$ tables

Figure 5.24: CPNs of the switch activity: the red crosses represent the canceled arc

Message triggering operator $Pick(\{M_i, S_i\}_{i \in I})$

$Pick$ triggers one sub process S_i by the arriving of a message $OnMessage(M_i)$ from partner PA_i, which is represented as the remote activation place $a^{in}_{PA_i}$. So $Pick$ operator is a combination of a set of $OnMessage$ activities (in figure 5.25b).

Parallel operator $flow(\{S_i\}_{i \in I})$

$Flow$ executes the activities S_i in parallel. It terminates when all the activities are finished (fork-join). So we add a^{in}, a^{out}, t^{in}, and t^{out} to compose the sub processes together in parallel (in figure 5.25c).

Conditional operator $If(\{(con_i(\overline{X_i}, \overline{V_i}), S_i)\}_{i \in I})$

If activity provides conditional behavior like $Switch$. It consists of an ordered list of one or more conditional branches defined by the if and optional $elseif$ elements, followed by an optional $else$ element. The figure 5.25d illustrates the CPN model of an if activity.

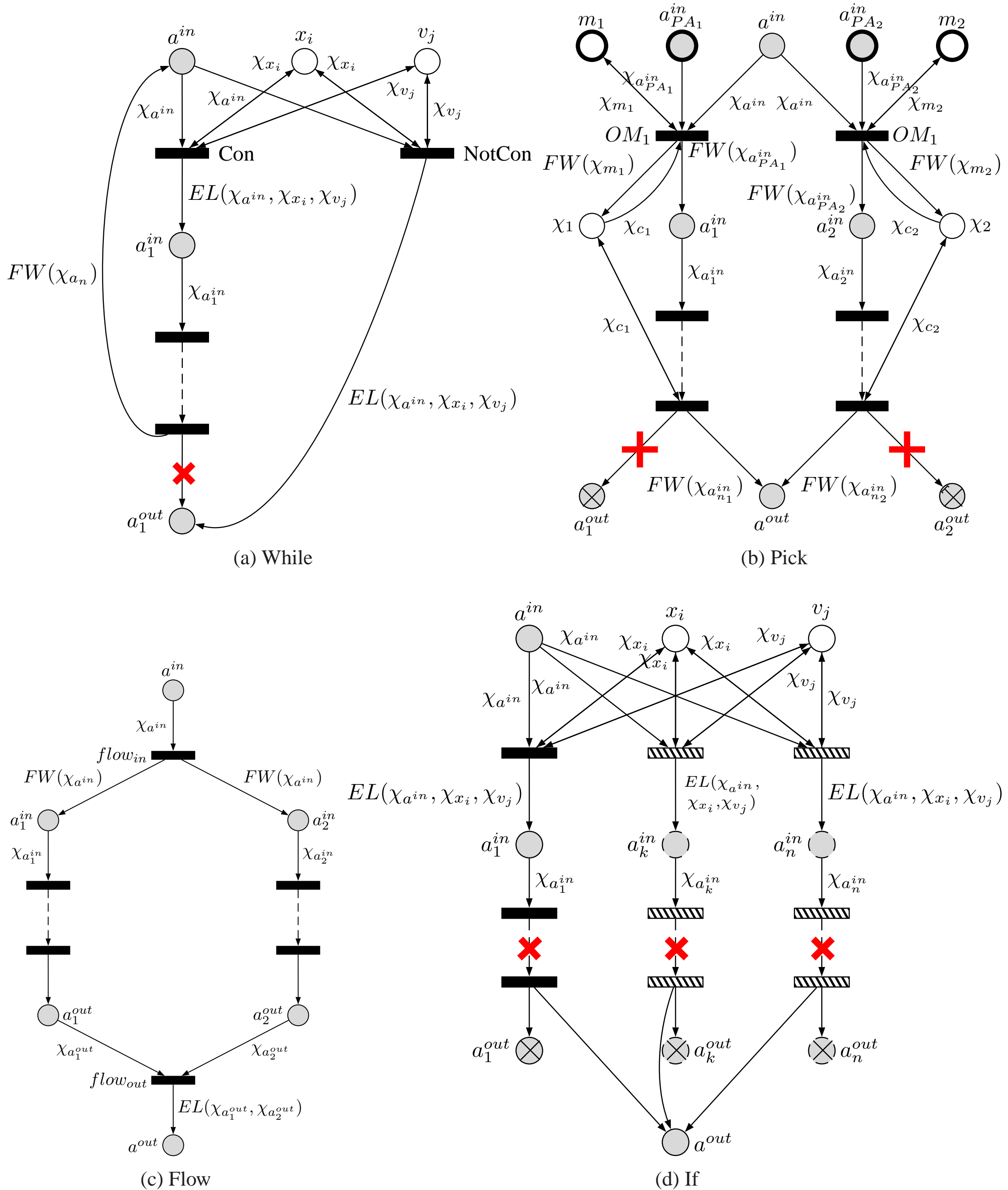


Figure 5.25: CPNs of the While, Pick, Flow, and If activities: the dashed-line places represent the optional places, the twill-style filled transitions represent the optional transitions, and the red crosses represent the canceled arcs

Conditional operator $RepeatUntil(\{(con_i(X_i, \overline{V_i}),S_i)\}_{i\in I})$

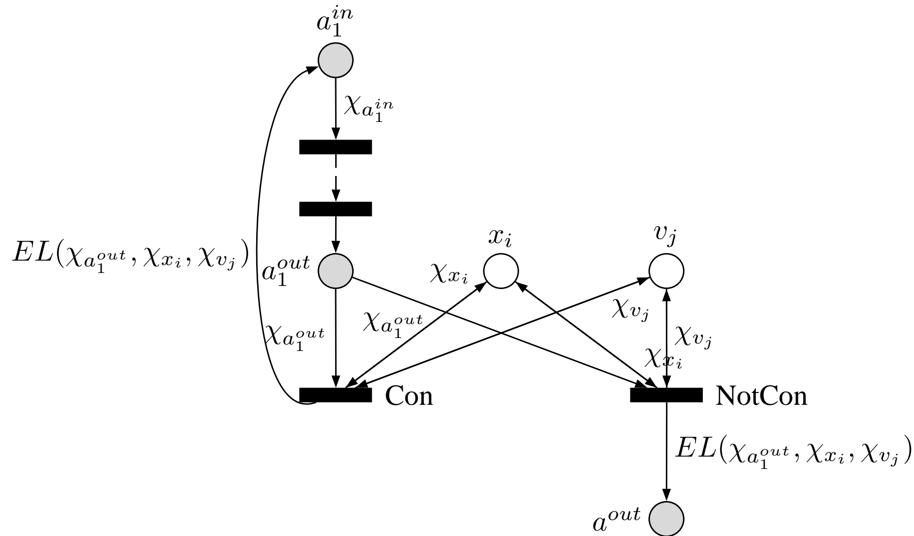


Figure 5.26: CPNs of the $RepeatUntil$ operator

The $RepeatUntil$ activity provides for repeated execution of a contained activity. The contained activity is executed until the given Boolean $condition$ becomes true. The condition is tested after each execution of the body of the loop. In contrast to the $while$ activity, the $repeatUntil$ loop executes the contained activity at least once. The figure 5.26 illustrates the CPN model of a $repeatUntil$ activity.

5.5.4 sub process with enclosed environment: Scope

A $Scope$ provides the context which influences the execution behavior of its enclosed activities, which can be arbitrary depth nested and structured. This behavioral context includes variables, partner links, message exchanges, correlation sets, and optional syntactic constructs, such as event handlers, fault handlers, a compensation handler, and a termination handler. Note that in WS-BPEL 2.0, $Invoke$ can be defined also like scope, which contains the optional syntactic constructs for event handling. In section 5.5.4, we will introduce first the handlers and then how to composite them in a scope.

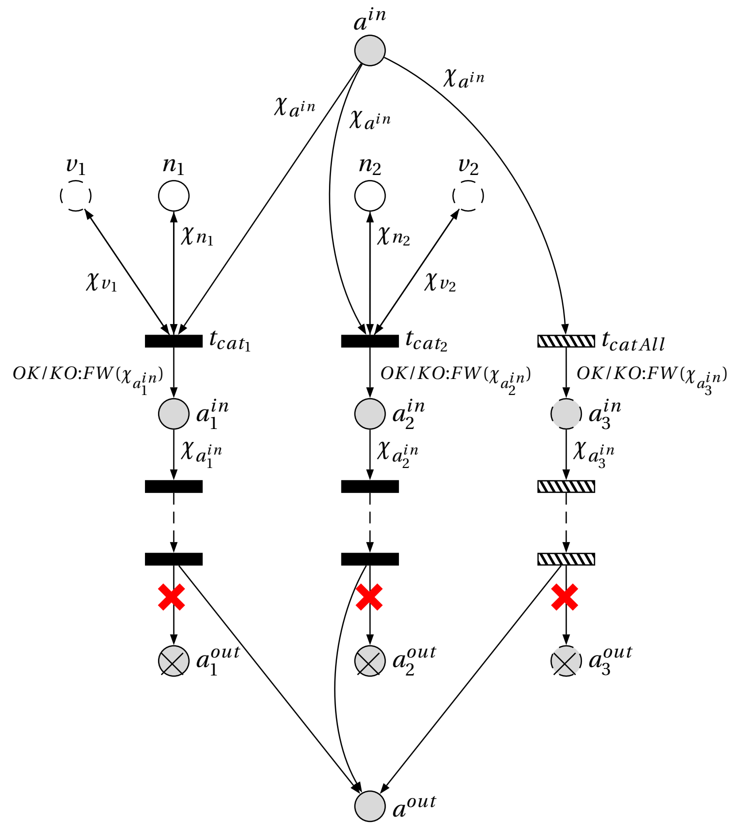


Figure 5.27: CPNs of the $faultHandlers$ activity which contains two $catch$ and one $catchAll$ activities: the dashed-line places represent the optional places, the crossed places represented the deleted places, the twill-style filled transitions represent the optional transitions, and the red crosses represent the canceled arcs. The $catchAll$ activity has no data input place

Fault handlers

Explicit fault handlers, if used, attached to a $scope$ provide a way to define a set of custom fault-handling activities, defined by $catch$ and $catchAll$ constructs. $Catch$ and $catchAll$ are the fault handling activities which correspond to $throw$ and $rethrow$. They can be defined in $faultHandlers$. Each $catch$ with the attributes "faultName", and/or "faultVariable". construct is defined to intercept a specific kind of fault. A $catchAll$ clause can be added to catch any fault not caught by a more specific fault handler. The figure 5.27 illustrates the CPN model of a $faultHandler$ activity. There are various sources of faults in BPEL. A fault response to an $invoke$ activity is one source of faults, where the fault name and data are based on the definition of the fault in the WSDL operation. A $throw$ or $rethrow$ activity is another source, with explicitly given name and/or data.

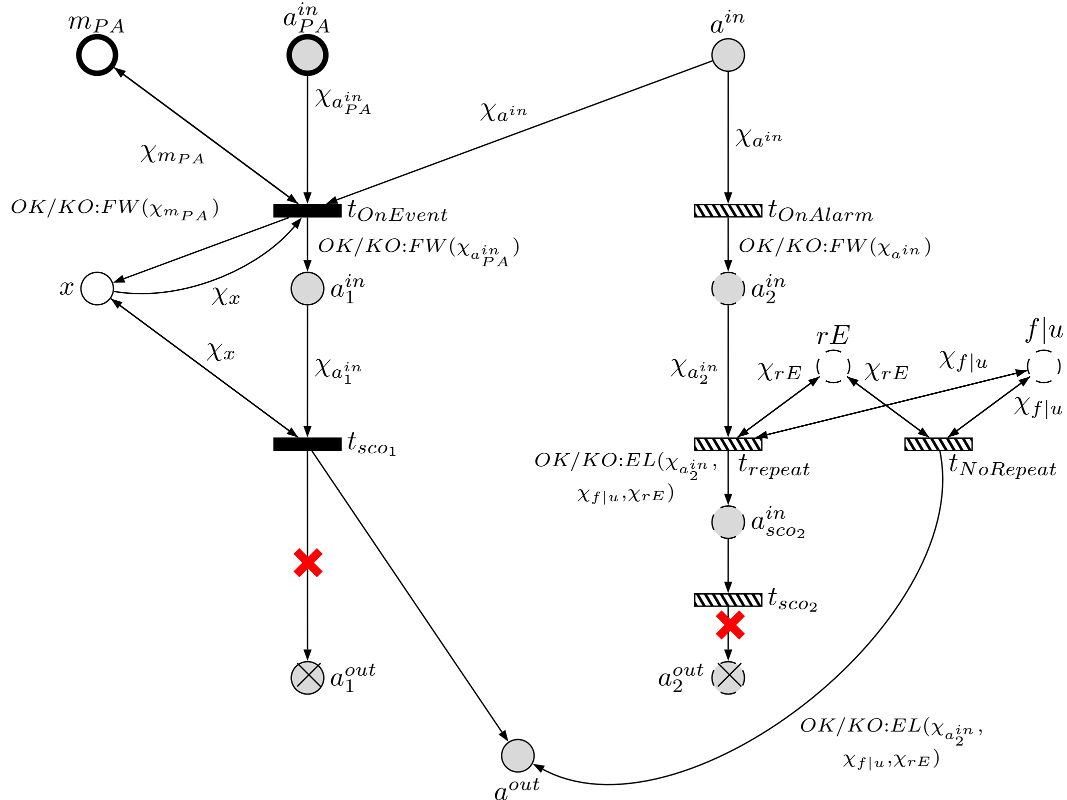


Figure 5.28: CPNs of the $eventHandler$ operator: place rE: $repeatEvery$, the alarm repeat interval; $f|u$: $for|unti$, the start or end point of alarm repeat. The dashed-line places represent the optional places, the crossed places represented the deleted places, the thick-line places represent the remote places, the twill-style filled transitions represent the optional transitions, the red crosses represent the canceled arcs.

$EventHandlers$

These event handlers can run concurrently and are invoked when the corresponding event occurs. The child activity within an event handler *must* be a $scope$ activity. There are two types of events. First, events can be inbound messages that correspond to a WSDL operation. Second, events can be alarms, that go off after user-set times. The figure 5.28 illustrates the CPN model of a $eventHandler$ activity.

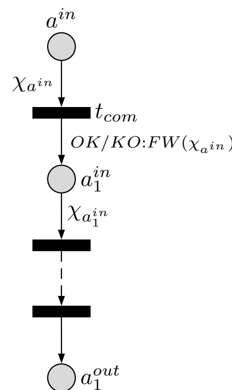


Figure 5.29: CPNs of the compensationHandler and TerminationHandler

$CompensationHandler$ **and** $TerminationHandler$

BPEL allows scopes to delineate that part of the behavior that is meant to be reversible in an application-defined way by specifying a compensation handler. Termination handlers provide the ability for scopes to control the semantics of forced termination to some degree. An arbitrary depth nested and structured activity is capsulated in the handlers. The figure 5.29 illustrates the CPN model of a $CompensationHandler$ or $terminationHandler$ activity. The $compensate$ activity is used to start compensation on all inner scopes that have already completed successfully, in default order. A compensation handler can be invoked by using the $compensateScope$ or $compensate$. The $compensateScope$ activity is used to start compensation on a specified inner scope that has already completed successfully. Both these two activities *must* only be used from within a fault handler (in $catch$ or $catchAll$), another compensation handler, or a termination handler.

Figure 5.30 represents the CPN model of $Scope$. Additional activation place $a_{throw}^{out\prime}$ is added to link the transition $throw$ with the corresponding $faultHandler$, place $a_{com}^{out\prime}$ is added to link the transition $compensate$ with the corresponding $compensationHalder$, and place $a_{sco}^{out\prime}$ is added to

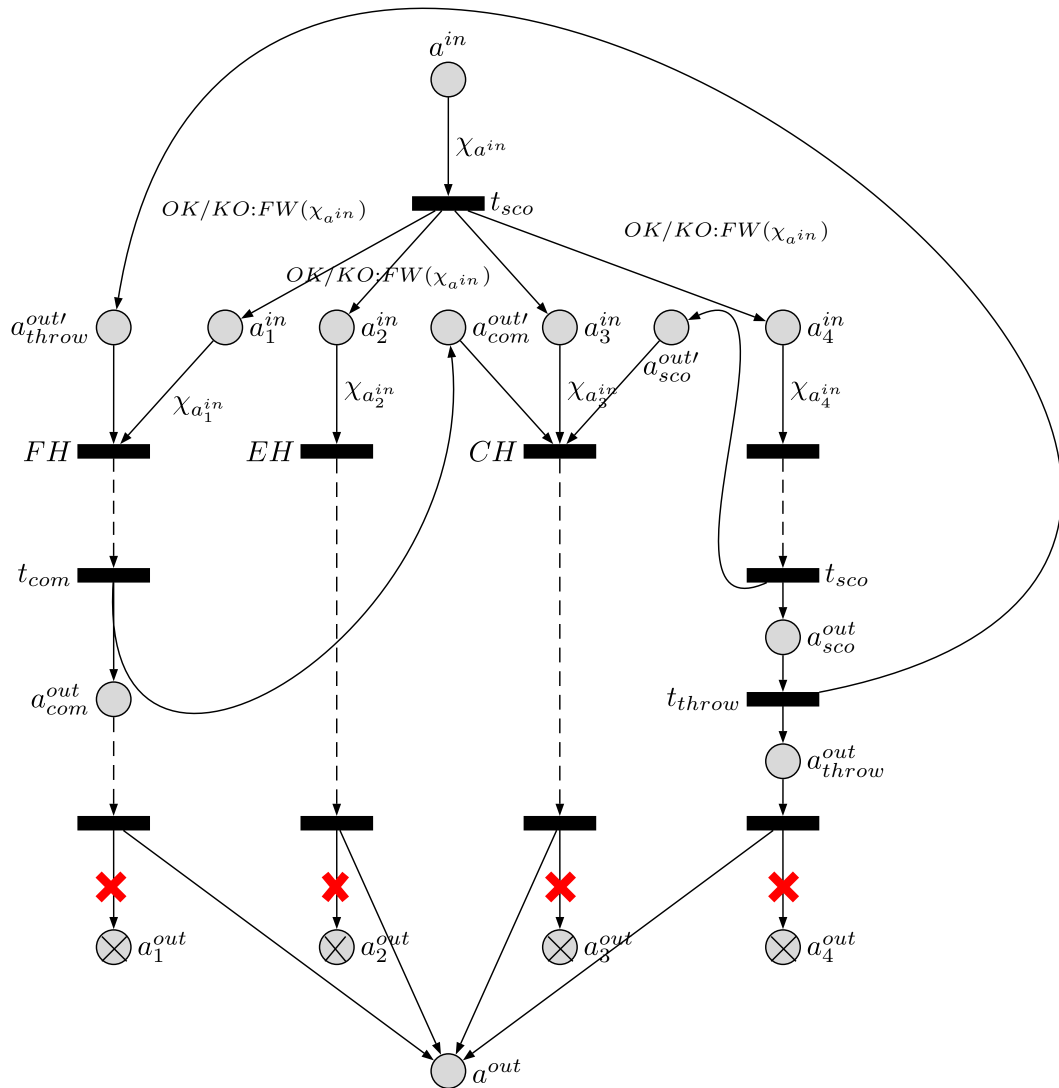
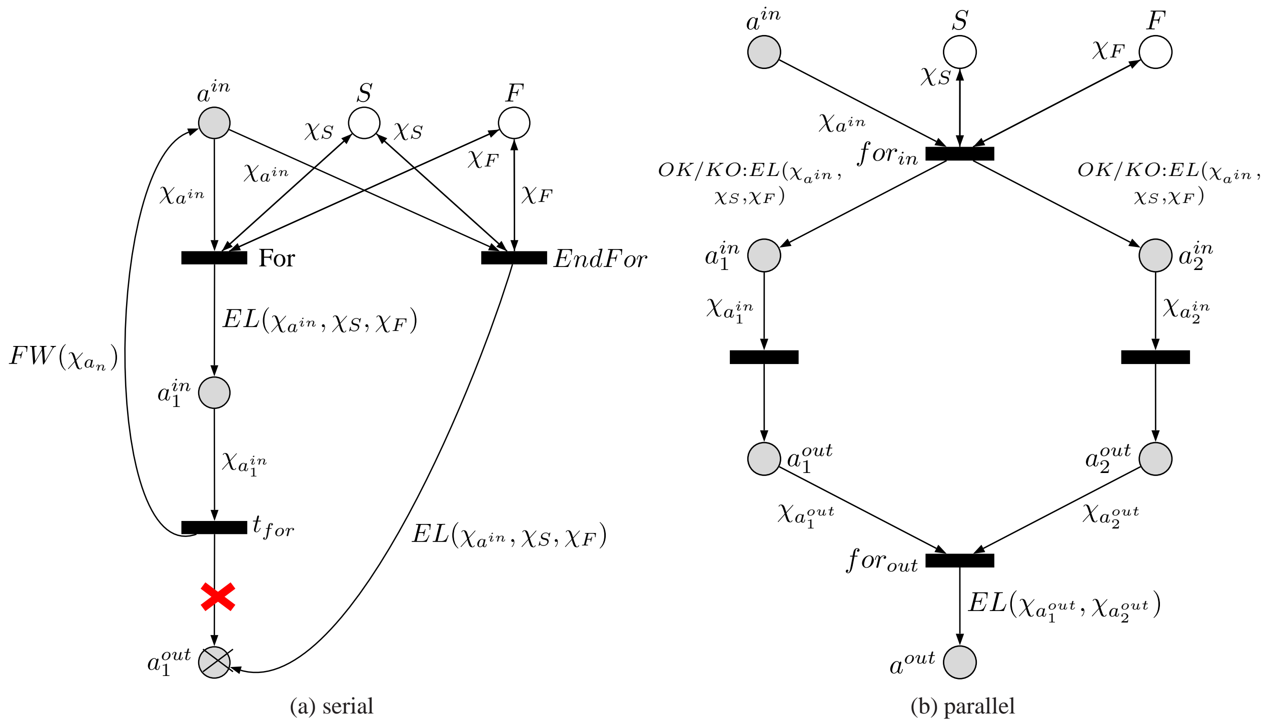


Figure 5.30: CPNs of the *scope* operator: the crossed places represented the deleted places, the red crosses represent the canceled arc. The *catchAll* activity has no data input place. Additional activation places $a^{out\prime}_{throw}$, $a^{out\prime}_{com}$, and $a^{out\prime}_{sco}$ link the handlers with the primary scope process

make sure the $compensationHalder$ only compensate the completely executed $scope$ before the $throw$ activity.



(a) serial (b) parallel

Figure 5.31: CPN model of forEach activity: the thick-line places represent the remote places

Conditional operator $ForEach(\{(con_i(X_i, V_i), S_i)\}_{i \in I})$

The $forEach$ activity will execute its contained $scope$ activity exactly N+1 times where N equals the $finalCounterValue$ minus the $startCounterValue$. The attribute $parallel$ defines if the $scope$ activity contained in $forEach$ should be executed in parallel or not. So $forEach$ should be modeled in two ways. The figure 5.31a and 5.31b respectively illustrate the CPN model of a $forEach$ activity in case of serial and parallel.

5.6 Case study: the CPN model of foodshop

Figure 5.32 illustrates a small BPEL service LocalSupplier which communicates with the BPEL service WAREHOUSE. It contains three basic activities in a sequence: a receive activity $Rec_W H_r equest$

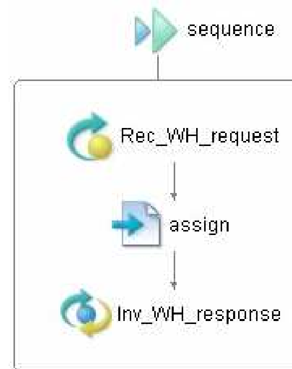


Figure 5.32: A small BPEL process LocalSupplier which offers order items for BPEL service WAREHOUSE

receive the item request from the WAREHOUSE, an assign activity ”prepares” the item (in a true() function), and an invoke activity $Inv_W H_r esponse$ confirms and ”sends back”the item to the WAREHOUSE. Figure 5.33 is its CPN model. The bolded activation places $cPInPNNet_PA$ (in gray color), represents the remote activation place for receive activity $Rec_W H_r equest$. The local activation place $cPInPNNet$ is represented as a place in gray color without bold. The bolded data places $msgP0$ and $msgP1$ represent the remote data places $Request/asyncData_PA$ and $Request/PID_PA$, which are the name of the requested item and the process ID. The data dependencies are illustrated on the arc of the corresponding output arc expressions. Then the assign activity $D2B0L2T4Assign$ assigns the data $true()$ and the order $Request/PID$ to variable $callbackResponse$ as $msgP7$ ($callbackResponse/callbackData$) and $msgP8$ ($callbackResponse/PID$) which are the input variables of the asynchronous invoke activity $D2B0L3T4Invoke$. So transition $D2B0L3T4Invoke$ has two activation places $D2B0L3cPOut4Invoke$ and $D2B0L2cPOut4Assign_PA$ to respectively represent the local and remote output activation places.

Figures 5.34, 5.35, and 5.36 illustrate the CPN model of the SHOP, WAREHOUSE, and SUPPLIER processes.

5.7 Related works

Petri nets have an intuitive graphical representation of BPEL processes and a brand of analysis tools could be used. In most of existing work, places are used to represent the system (a BPEL process) states and transitions are used to represent activities. The Petri nets models of BPEL services are

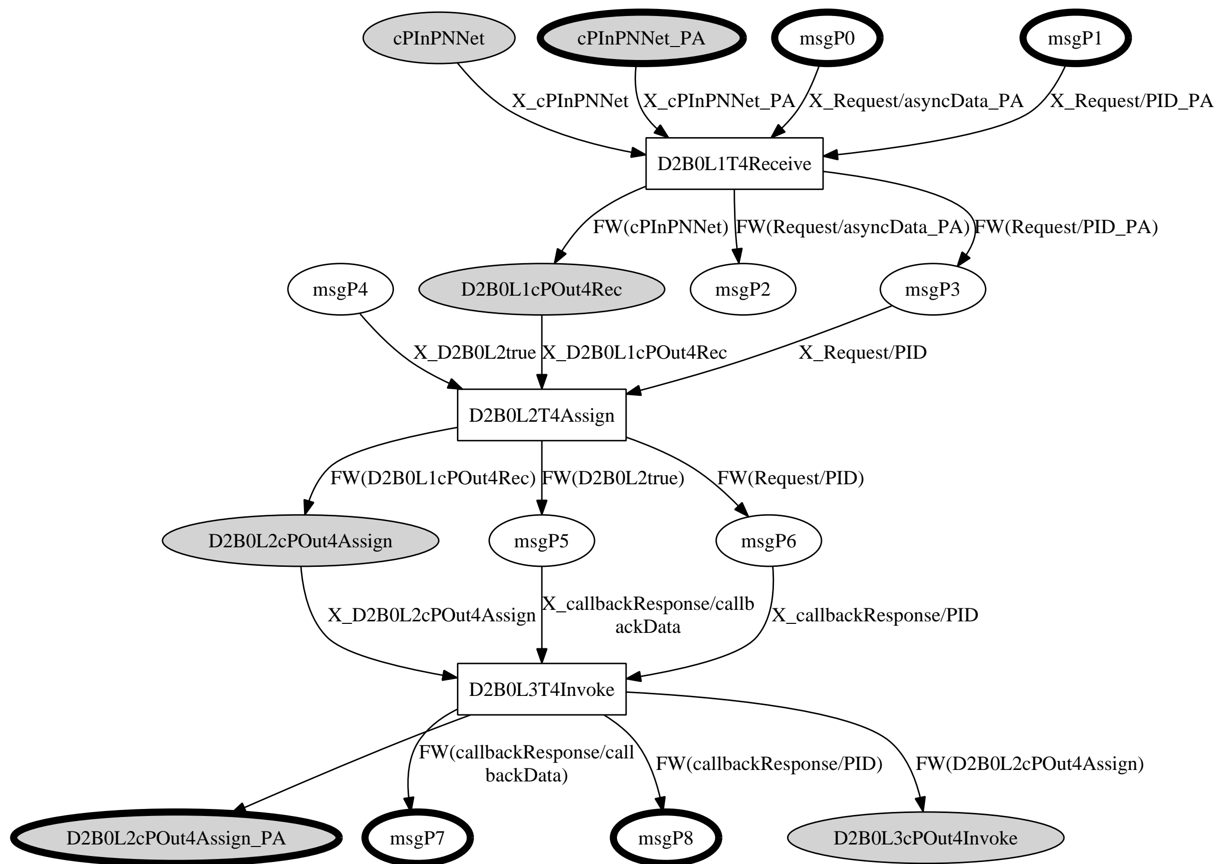


Figure 5.33: The CPNs model of LocalSupplier

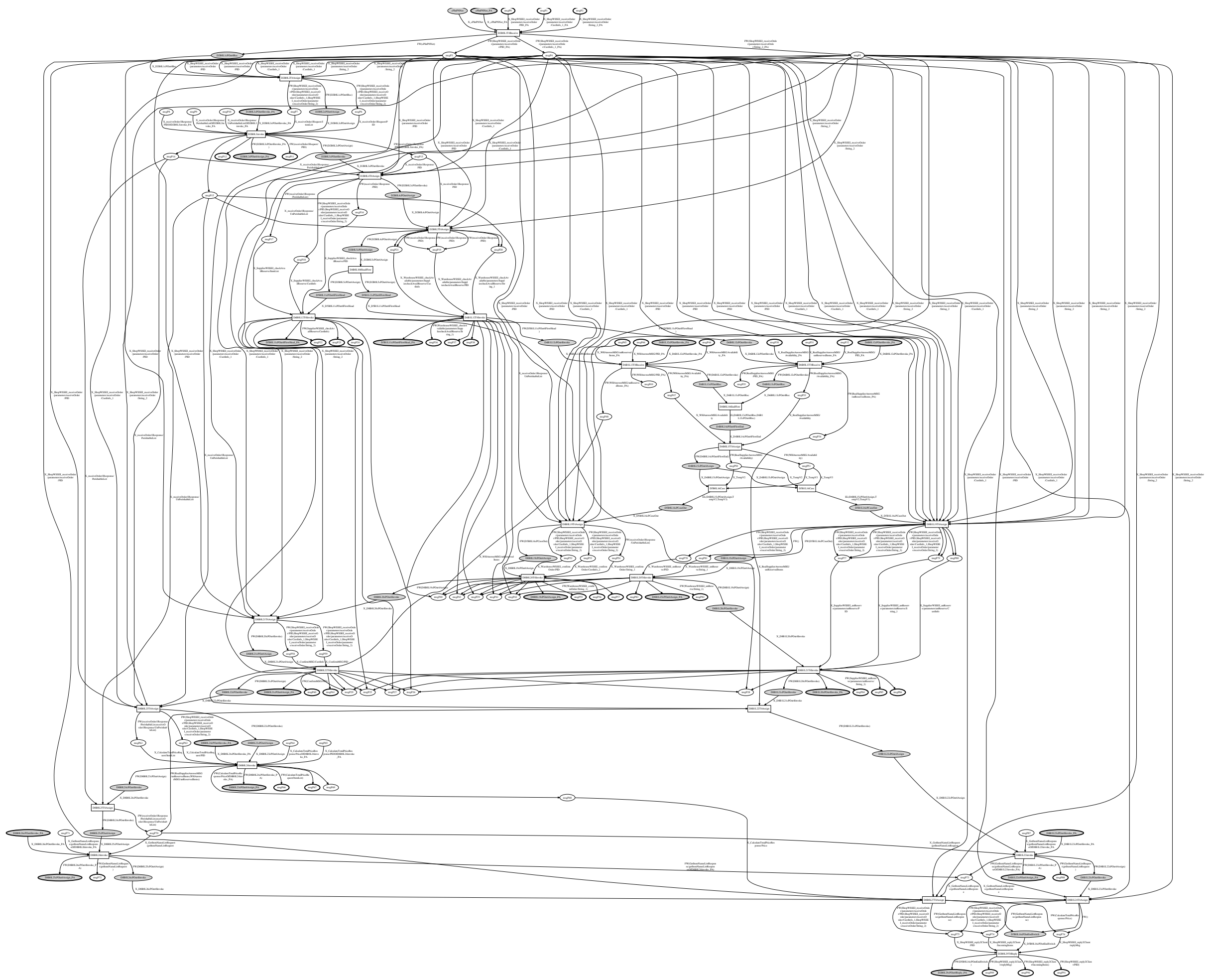


Figure 5.34: The CPNs model of SHOP process

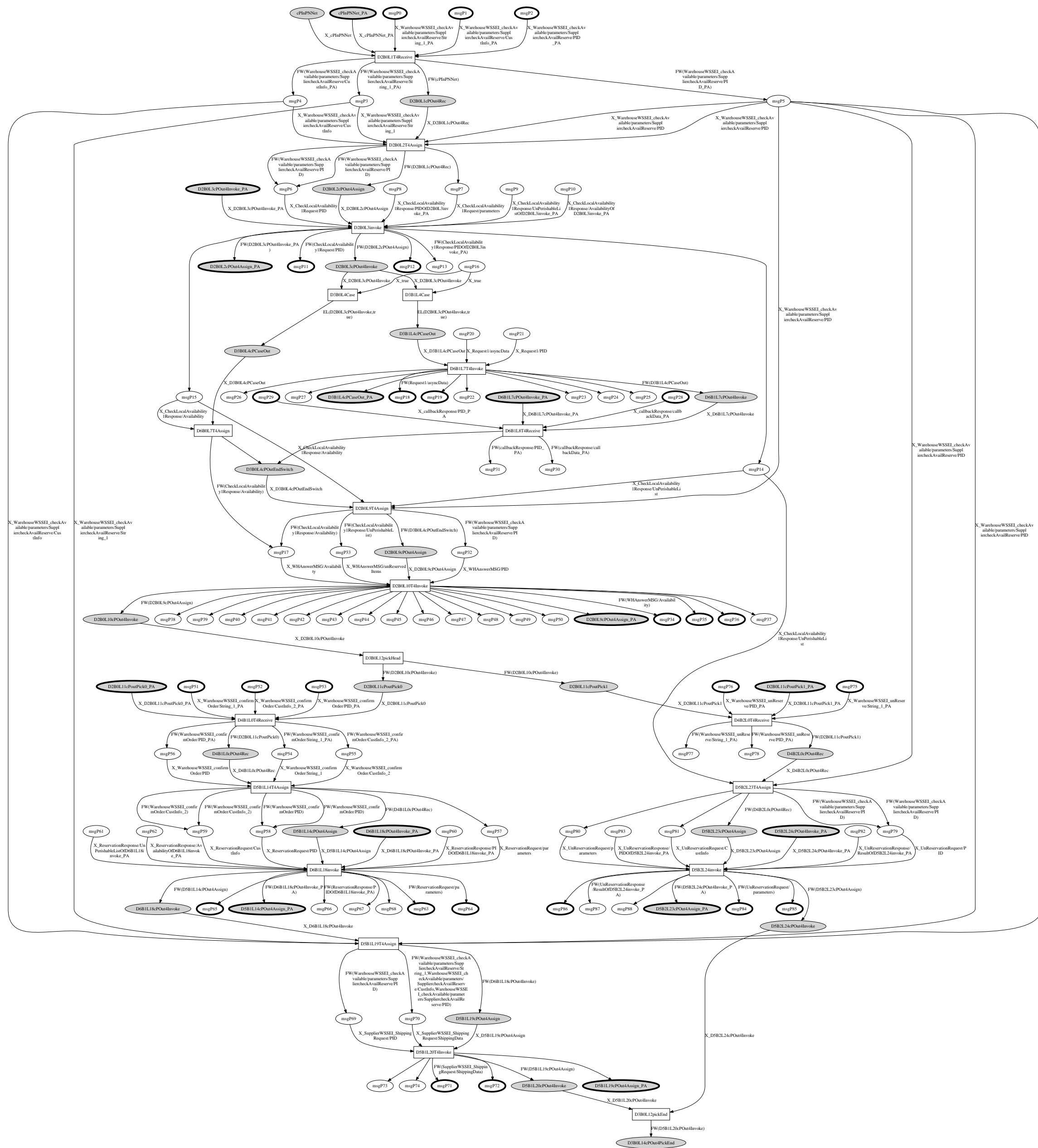


Figure 5.35: The CPNs model of WAREHOUSE process

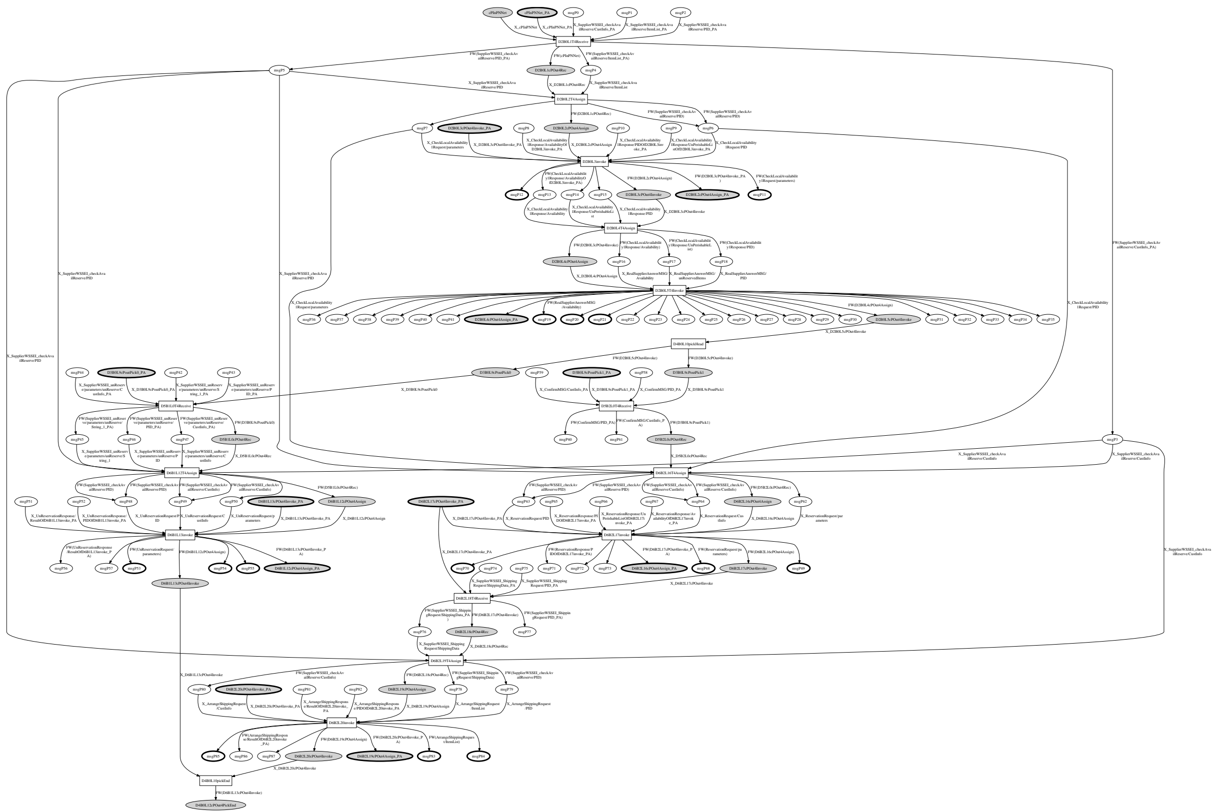


Figure 5.36: The CPNs model of the SUPPLIER process

usually used for the composition verification, validation, feasibility checking, dead path elimination, and controlability criterion checking [92, 117, 48, 85, 52, 79, 80].

There are also some work modeling BPEL services with high level Petri nets [143, 150]. [143] represents Web service composition description languages with CPN in terms of WSCI (Web Service Choreography Interface) for Web service composition verification. [150] models Web service with CPN to verify the closure, availability, and security of Web service.

Modeling BPEL services as algebra processes is for orchestration and choreography study [133, 114, 38], dead-path-elimination [132], and ambiguity behavior verification [45] of Web service. We refer the reader to [10] and [140] for the surveys of formal methods of for Web services modeling.

For monitoring and diagnosing a DES which is modeled with Petri nets, the major method is to detect all the reachable trajectory according to ordered observations ([8, 96, 110]) which suffers from the state explosion problem. [144] proposes a decentralized model-based diagnosis algorithm based on the PNs model in [78] by inferring backward along the data dependency paths. But in [144], the diagnosis algorithms for local BPEL process does not support processes contain loops.

There are some other works using different models to do diagnosis. [20] proposed a CPN model and unfolding algorithm for the supervision and diagnosis of Web service, but focused on the change of the system components (which are modeled as tokens) and it did not offer the direct translation. In [24], a system is modeled with process algebra containing faulty behavior models. The diagnosis is to compare all possible action traces with the observations. All the faulty actions on the matched traces are the diagnosed faults. [141] models BPEL services as synchronizing automata pieces, and builds the behavioral models from the process description.

Chapter 6

Decentralized architecture for CPN based diagnosis

6.1 Introduction

The system distribution offers an effective solution for large complex systems because of its advantages such as, resource sharing, openness, concurrency, scalability, fault tolerance, and transparency. Meanwhile the maintenance of the distributed systems, including diagnosis, is much more demanding.

According to the distribution topologies that we discussed in chapter 2 section 2.5, two architectures, decentralized and distributed, should be fully studied when the Web services applications are concerned. The decentralized architecture [33, 97, 34, 98, 13] is more effective for the Web service applications located on different hosts but the interface data is not confidential. While the distributed architecture [102, 124, 56, 125, 59, 37, 118, 57] is more suitable for those applications, of which the authorizations of the components are limited.

In this chapter, we discuss the decentralized diagnosis, the communication protocols and diagnosis algorithms are given. The global consistency of the decentralized diagnosis is proposed, which is inspired by [149]. The same example of the dining philosophers will be discussed in a distributed architecture: each philosopher is represented as a system component as illustrated in figure 6.1.

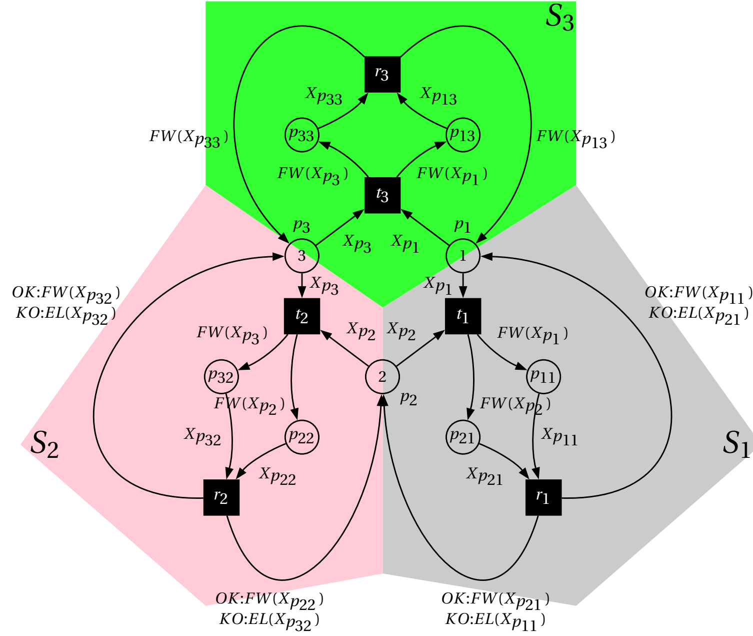


Figure 6.1: Dining philosopher: as three distributed parts S_1, S_2, S_3

6.2 Decentralized system

The decentralized diagnosis architecture is inspired by [5] and [144]. We consider a system composed by a set of interacting software components inside a system. The CPN models (see figure 6.2) can be seen as a set of CPNs (a CPN for each component), which share a set of places called *bordered places* (see definition 70). These place-bordered CPNs are called CPN partnership (see definition 71).

Definition 70 (Bordered places set)**.** *Let $\mathcal{N}$ be a set of CPNs we range over using N_i, we define the following notations:*

1. $RP_i = P_i \cap \bigcup_{j \neq i} P_j$ *is the set of bordered places of N_i;*
2. $RP_{ij} = RP_i \cap RP_j$ *is the set of bordered places between N_i and N_j;*
3. $RP_i^{in} = \{rp \in RP_i | P^{\bullet} \subseteq T_i \wedge^{\bullet} P \cap T_i = \emptyset\}$ *is the set of input bordered places of N_i;*
4. $RP_i^{out} = \{rp \in RP_i | P^{\bullet} \cap T_i = \emptyset \wedge^{\bullet} P \subseteq T_i\}$ *is the set of output bordered places of N_i;*
5. $RP_{i \to j}^{in} = RP_i^{in} \cap RP_j^{out}$ *is the set of input bordered places between N_i and N_j;*

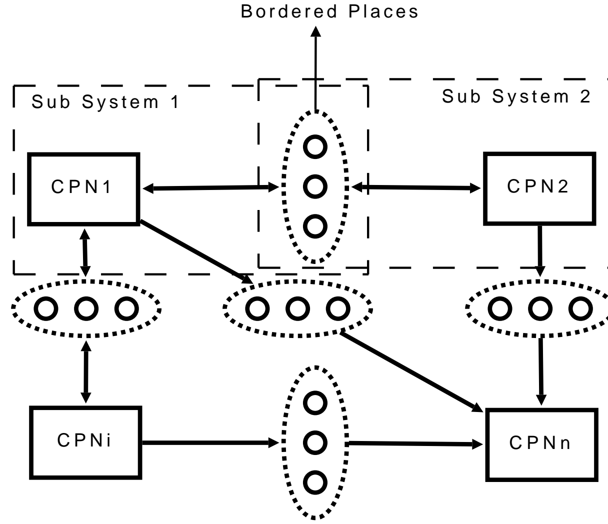


Figure 6.2: Place-bordered CPN model in a decentralized architecture

6. $RP^{out}_{i\to j} = RP^{out}_i \cap RP^{in}_j$ *is the set of output bordered places between* N_i *and* N_j.

Note $\{p|p \in P_i \setminus RP_i\}$ are the inner places set of N_i and for $i \to j$, we call N_i the source CPN and N_j the target CPN.

Definition 71 (CPN Partnership)**.** *Let* $\mathcal{N}$ *be a set of CPNs* N_i, *we call* $\mathcal{N}$ *a* CPN partnership *iff*

- $RP^{in}_i \cup RP^{out}_i = RP_i$;
- $\forall i, j \in N$, $RP^{in}_i \cap RP^{in}_j = \emptyset \wedge RP^{out}_i \cap RP^{out}_j = \emptyset$.

Each N_i *is a partner in* $\mathcal{N}$.

6.3 Diagnosis problem of decentralized system

The software system (seen as a whole large CPN) can be diagnosed with the approach proposed in chapter 4 in a centralized way. While sometimes, with the authority limit of some components, it is not allowed to get the whole CPN for diagnosis. So it is necessary to extend the existing centralized diagnosis approach to such kind of system. The idea is to locate a diagnoser in each component and to add a specific coordinator to handler the cooperation between the local diagnosers (see figure 6.3). To avoid the large amount of communications between the components of the components of the software systems, the coordinator keeps only the bordered places information of the partners to

coordinate the necessary communication between the local diagnosers. A local diagnosis component takes the local CPN model and local observation to generate the local diagnostics for the local recovery component. Each local diagnosis component processes locally.

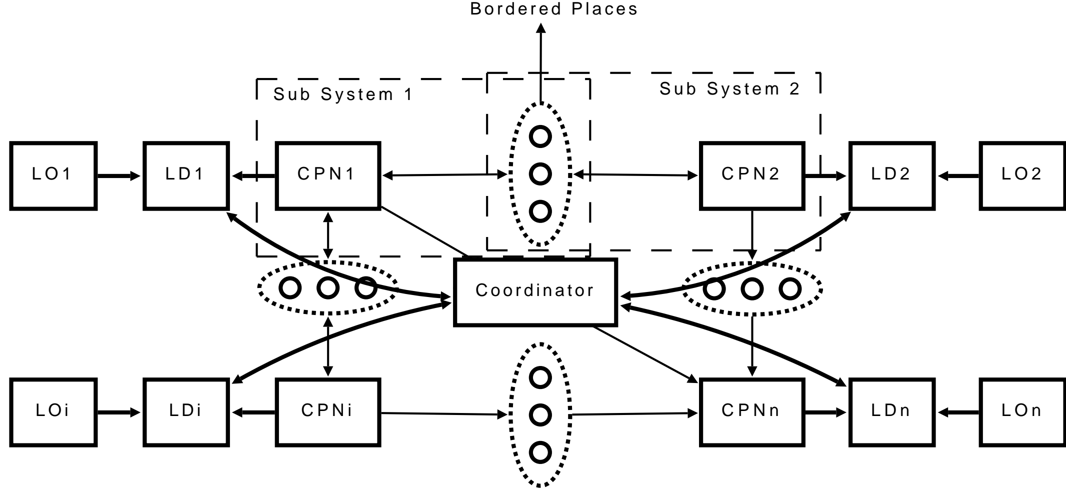


Figure 6.3: Decentralized diagnosis architecture of a set of place-bordered CPNs: LD represents the local diagnoser, LO represents the local observation.

The local diagnosis problem can adapted as $\mathcal{LD}_i$=$\langle M_{0_i}, (S(\mathcal{T})_i, \lhd_i), M_i\rangle$ according to the definition of the centralized diagnosis problem (see definition 54 in chapter 4). M_i represents a final marking on a local site, but it is not necessary to be a symptom marking. But the bordered places cannot have conflict markings in two communicating CPN model (see assumption 1), otherwise this conflict is ignored. More specifically, the marking of the target CPN *covers* the marking of the source CPN.

Proposition 1. *Given a CPN partnership $\mathcal{N}$, its marking $\mathcal{M} = \{M_i\}$ satisfies the constraint:* $\forall rp \in RP^{in}_{i\to j} = RP^{out}_{i\to j}$, $M_j(rp) \succcurlyeq M_i(rp)$.

This proposition means that during the communication of two partner CPNs, the marking of the bordered places set of the target CPN model always covers that of the source one. This proposition ensures that we can decompose a CPN diagnosis problem into its CPN partnership diagnosis problems.

Proof. The proof is obvious based on the definitions of partnership 71 and covering relation 55 because $\forall rp \in RP^{in}_{i\to j} = RP^{out}_{i\to j}$, rp is just the same place which is the border of two CPNs. □

Then we can define the decentralized diagnosis problem as follows:

Definition 72 (Decentralized CPN diagnosis problem)**.** *Given a CPN partnership $\mathcal{N}$, each CPN partner N_i has a local diagnosis problem $\mathcal{LD}_i$, a decentralized diagnosis problem of $\mathcal{N}$ is $\mathcal{D} = \{\mathcal{LD}_i | \exists ! \hat{i} \in I, \exists p \in P_{\hat{i}}, M_{\hat{i}(p)=r}\}$.*

Definition 73 (Decentralized CPN diagnosis)**.** *Given a CPN diagnosis problem $\mathcal{D}$, a decentralized diagnosis $Diag(\mathcal{D})$ is defined as $\exists \mathcal{D}' = \{LD'_i\}$, s.t., for each $LD'_i = \langle M'_{0_i}, (S(\mathcal{T})'_i, \lhd'_i), M'_i \rangle$, :*

(i) $M_{0_i} = M'_{0_i}$, $S(\mathcal{T})'_i = S(\mathcal{T})_i$, $\lhd'_i = \lhd_i$, $M_i \not\succcurlyeq M'_i$;

(ii) $\forall rp \in RP^{out}_{i \to j} = RP^{in}_{i \to j}$, $M'_j(rp) \not\succcurlyeq M'_i(rp)$.

So $Diag(\mathcal{D}) = \bigcup\limits_{i \in I} Diag(LD_i) \setminus \bigcup\limits_{i \in I} RP_i$.

3-d-p example 24. *Consider the scenario in figure 6.1, we can solve the global diagnosis problem by solving three local diagnosis problems: LD_1, LD_2, and LD_3, each of which has its own local observation and initial, final markings. Note the bordered places for each two partners, which are places p_i, their final markings $M'_j(p_i) = 0$ which satisfy condition (ii) of the decentralized diagnosis 73.*

6.4 Diagnosis approach

According to the assumption 1, symptom occurs on one local site, then the local diagnoser is triggered. When the local diagnoser needs to communicate with the partner(s), it sends request(s) through the coordinator. This process continues when the coordinator confirms there is no more diagnosis request to proceed. Then the coordinator integrates the diagnosis results by substituting the intermediate diagnosis results backward. In this section, we explain in detail the protocols and algorithms of the local diagnosers and the coordinators

6.4.1 Diagnosis protocol

A request (or response) represents the information emitted from a local diagnoser to the coordinator (or from the coordinator to a local diagnoser). The local diagnosers communicate with the coordinator through sending and receiving requests and responds, which are queued separately. So both a local diagnoser and the coordinator have two queues: one for requests, and the other for responds. Both the local diagnosers and the coordinator handle the calculation of impossible diagnosis solutions (name as counter-diagnosis for convenience) and diagnosis requests and responds in turn.

To distinguish the different instances of the same diagnoser and the different local diagnosers, the requests and responses contain at least the information of its local diagnoser id, instance id, the counter-diagnosis/fault flag, the bordered places, and the faulty transition modes. In this section, we define the workflow and the communicating protocols between the local diagnoser (figure 6.4) and the coordinator (figure 6.5).

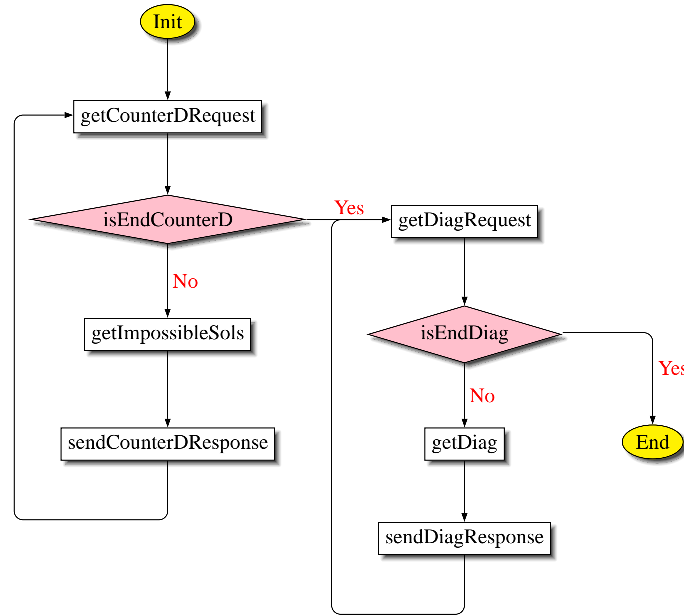


Figure 6.4: The flowchart of the local diagnoser: $getImpossibleSols$ and $getDiag$ are local counter-diagnosis and diagnosis generation functions which are explained in chapter 4.

A local diagnoser is initiated by the symptom detected on its host local component and keeps alive waiting for the requests from the coordinator. In first case, when one or more than one exception is caught, the local diagnoser performs the diagnosis algorithm to generate one primary counter-diagnosis set by function $getImpossibleSols$ (and one diagnosis set by function $getDiag$) which contains the local bordered places that need to infer further and the local counter-diagnosis (and faulty transitions modes). Then the local diagnoser sends the counter-diagnosis and the primary diagnosis result together to the coordinator as the first diagnosis request. In the second case, the local diagnoser keeps active during the inferring process until it receives a terminating request from the coordinator. Then it starts the diagnosis process and keeps activate until it receives another terminating request from the coordinator. Figure 6.4 illustrates the flowchart of the local diagnoser.

When the coordinator receives this request, it first handles the counter-diagnosis, and then the primary diagnosis result. According to the bordered places information, the coordinator invokes the

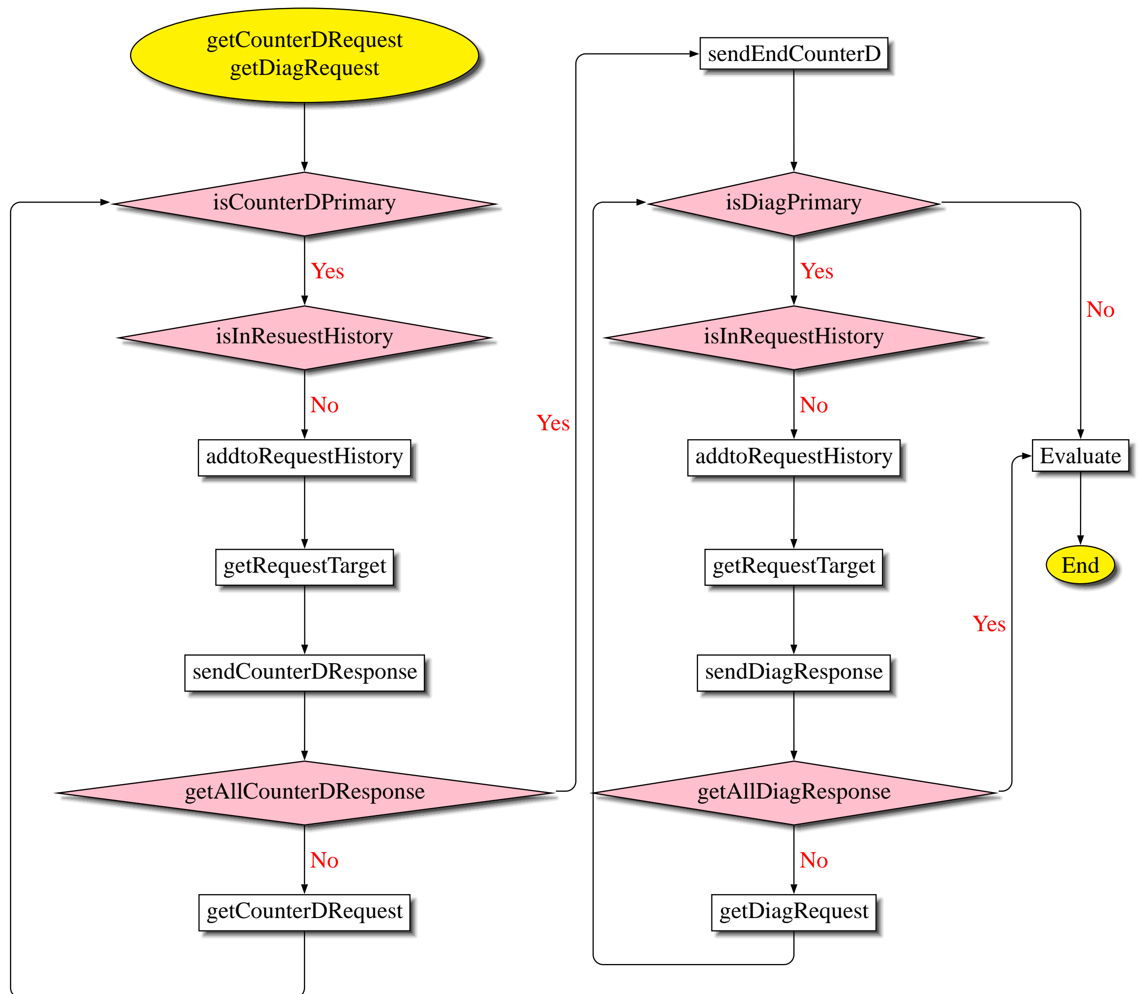


Figure 6.5: The flowchart of the coordinator: $isCounterDPrimary$ and $isDiagPrimary$ mean if the request needs to be inferred further; $isInCounterDHistory$ and $isInDiagHistory$ mean if the request is received before; $getRequestTarget$ finds the corresponding local diagnoser to send request for further inferring; $Evaluate$ evaluates the first primary diagnosis/counter-diagnosis with the further diagnosis/counter-diagnosis result and calculate the global diagnosis.

local diagnosers of the source CPNs to infer their own local counter-diagnosis, waits and records the responds. If there are the bordered places to handle in the respond, the coordinator calls their corresponding local diagnosers of the source CPNs. To avoid redundant request, the coordinator records all the requests, when it receives a request existed in the history, it does not handle it. The counter-diagnosis inferring process terminates when no new counter-diagnosis in the local diagnosis response. The coordinator sends a terminating signal to each local diagnoser to inform them. The coordinator proceeds with the primary diagnosis requests in the same routine.

The last step, the coordinator substitutes the bordered places in the primary diagnosis and counter-diagnosis with the further solved result, which we call it "evaluate", and computes the minimal diagnosis with the operator $\overset{\cup}{\times}$.

6.4.2 Diagnosis algorithm

Algorithm 5 $LDS(\{p_i\})$: local diagnosis for $Q_{\hat{M}}$

Input: $\langle CFFlag, PSet\rangle$: a counter-diagnosis/fault places flag and a places set as symptoms;
Output: D: local counter-diagnosis/diagnosis (to solve further);

1: **if** $\langle CFFlag, PSet\rangle$ is from local **then**
2: $CounterD = getImpossibleSols(Q_{\hat{M}})$;
{calculate the counter-diagnosis (see algorithm 2 in chapter 4)}
3: $D = getDiag(Q_{\hat{M}})$;
{calculate the diagnosis (see algorithm 4 in chapter 4)}
4: **return** D ;
5: **else if** $\langle CFFlag, PSet\rangle$ is not $ENDCounterD$ or $ENDDiag$ **then**
6: $Q_{\hat{M}} = getInequSys(CFFlag, PSet, Q_{\hat{M}})$; {update the inequations system, algorithm 6}
7: **if** $CFFlag = C$ **then**
8: $CounterD = CounterD \cup getImpossibleSols(getInequSys(CFFlag, PSet, Q_{\hat{M}}))$;
9: **else if** $CFFlag = F$ **then**
10: D=$getDiag(Q_{\hat{M}})$;{DO NOT execution line 16}
11: **end if**
12: **else if** $\langle CFFlag, PSet\rangle$ is $ENDCounterD$ **then**
13: **return** $CounterD \cap RP^{in}$;
14: **else if** $\langle CFFlag, PSet\rangle$ is $ENDDiag$ **then**
15: **return** $D \cap RP^{in}$;
16: **end if**

The decentralized diagnosis algorithm consists the local diagnoser algorithm LDS (see algorithm 5) and the coordinator diagnoser algorithm GDS (see algorithm 7). They both pop a request from the $InQueue$ and push the response to the $outQueue$.

Algorithm LDS illustrates the process of the local diagnoser after popping a request from the $InQueue$ and before pushing the responses to the $outQueue$. According to the information contained in the $InQueue$, three situations are considered:

- if the request is from the local monitoring component, compute a primary counter-diagnosis and a primary diagnosis based on the current inequations system (line 1 to line 4);
- if the request is a counter-diagnosis request for further solving, compute a local counter-diagnosis based on the updated inequations system (line 7 to line 8);
- if the request is a diagnosis request, compute a local diagnosis based on the counter-diagnosis and then updated the inequations system (line 9 to line 10).

Algorithm 6 $getInequSys$: update the inequations system with new information

Input: $CFFlag$: $CFFlag = C/F$ means $PSet$ is counter-diagnosis/faults;
$PSet$: a places set;
$Q_{\hat{M}}$: current inequations system;
Output: $Q_{\hat{M}}$: the updated inequations system;
1: **if** $CFFlag = C$ **then**
2: **ForEach** $p \in PSet \cap P$ **do**
3: **if** $Eq_p \in Q^*_{\hat{M}}$ **then**
4: $r(Eq_p) = r(Eq_p) + b - l(Eq_p)$;
5: $l(Eq_p) = b$;
6: **else if** $Eq_p \in Q^r_{\hat{M}}$ **then**
7: Exit; {ignore the conflict information}
8: **end if**
9: **end for**
10: **else if** $CFFlag = F$ **then**
11: **ForEach** $p \in PSet \cap P$ **do**
12: **if** $Eq_p \in Q^*_{\hat{M}}$ **then**
13: $r(Eq_p) = r(Eq_p) + r - l(Eq_p)$;
14: $l(Eq_p) = r$;
15: **else if** $Eq_p \in Q^b_{\hat{M}}$ **then**
16: Exit; {ignore the conflict information}
17: **end if**
18: **end for**
19: **end if**
20: **return** $Q_{\hat{M}}$;

Algorithm 7 GDS: global diagnosis solution

Input: $CounterD_0$: a set of primary counter-diagnosis;
D_0: a set of primary diagnosis;
Output: $Diag$: the global integrated diagnosis set;
1: add each bordered place p of the element of $CounterD_0$ into the $outQueue$;
2: $CounterDList.add(CounterD_0)$; $\{CounterDList$ the list of counter-diagnosis to solve$\}$
3: add each bordered place p of the element of D_0 into the $outQueue$;
4: $DList.add(D_0)$; $\{DList$ the list of diagnosis to solve$\}$
5: **while** $inQueue \neq \emptyset$ **do**
6: pop out a response from $inQueue$;
7: add the response to $CounterDList$ or $DList$ as the further inferring result of request concerns p;
8: update $requestHistory$ concerns p;
9: **end while**
10: **if** all counter-diagnosis request are responded **then**
11: **ForEach** $i \in I$ **do**
12: $outQueue.add(i, ENDCounterD)$; $\{$terminate the counter-diagnosis inferring on the LDs$\}$
13: **end for**
14: **end if**
15: **if** all diagnosis request are responded **then**
16: **ForEach** $i \in I$ **do**
17: $outQueue.add(i, ENDDiag)$; $\{$terminate the diagnosis inferring on the LDs$\}$
18: **end for**
19: $Diag = Evaluate(CounterDList, DList)$;
20: **end if**
21: **return** $Diag$;

Function $getInequSys$ is called by LDS for regrouping the inequations of $Q_{\hat{M}}$ according to counter-diagnosis/places flag $CFFlag$ during the life cycle of the local diagnoser. For each unknown-token ($*$) inequation labeled by a place of the input place set, the inequation is adjusted and removed to either the black-token (line 1-9 of algorithm 6) or red-token (line 10-18 of algorithm 6) inequations sub set. To adjust an inequation, is to make sure the marking on the left side of the inequation to be b (line 1-5 of algorithm 6)(or r (line 10-14 of algorithm 6)) without violating the covering relation of the inequation by add a residual on the right side of the inequation.

The coordinator algorithm (GDS in algorithm 7) proceeds in two steps: first, to coordinate the counter-diagnosis and diagnosis requests by sending the requests (and receiving the responds) to the local diagnosers of (and from) the source (and target) CPNs of the bordered places (line 1 to line 9); second, to substitute the responds of further inferring request backward and integrate them to get the final diagnosis (function $Evaluate$ listed in the algorithm 8).

Algorithm 8 $Evaluate$: to evaluate the diagnosis/counter-diagnosis result and calculate the global diagnosis

Input: $CounterDList$: the list of the primary counter-diagnosis and the further inferred counter-diagnosis; $DList$: the list of the primary diagnosis and the further inferred diagnosis;
Output: $Diag$: the integrated diagnosis;

1: $Diag_p = DList.pop$;$\{Diag_p$: the further explain concern symptom bordered place $p\}$
2: **while** $Diag_p \neq Diag_0$ **do**
3: **ForEach** $Diag \in DList$ **do**
4: replace all $p \subseteq Diag$ with $Diag_p$;
5: **end for**
6: $Diag_p = DList.pop$;
7: **end while**
8: calculate the counter-diagnosis set $CounterDList$ in the same way;
9: $Diag = \overset{\cup}{\times} diag_i \setminus CounterDList$
10: **return** $Diag$;

6.4.3 Example: dining philosophers

3-d-p example 25. *Suppose the 3 dining philosophers are located on three local sites, S_1, S_2, and S_3, then there are three local bordered places p_1, p_2, and p_3. For S_i, $RP_i^{in} = RP_i^{out} = \{p_j, p_k\}$ with $i \neq j \neq k$ and $i, j, k \in \{1, 2, 3\}$. Furthermore, $RP_{1\to2}^{in} = RP_{1\to2}^{out} = RP_{2\to1}^{in} = RP_{2\to1}^{out} = \{p_2\}$; $RP_{2\to3}^{in} = RP_{2\to3}^{out} = RP_{3\to2}^{in} = RP_{3\to2}^{out} = \{p_3\}$; $RP_{1\to3}^{in} = RP_{1\to3}^{out} = RP_{3\to1}^{in} = RP_{3\to1}^{out} = \{p_1\}$.*

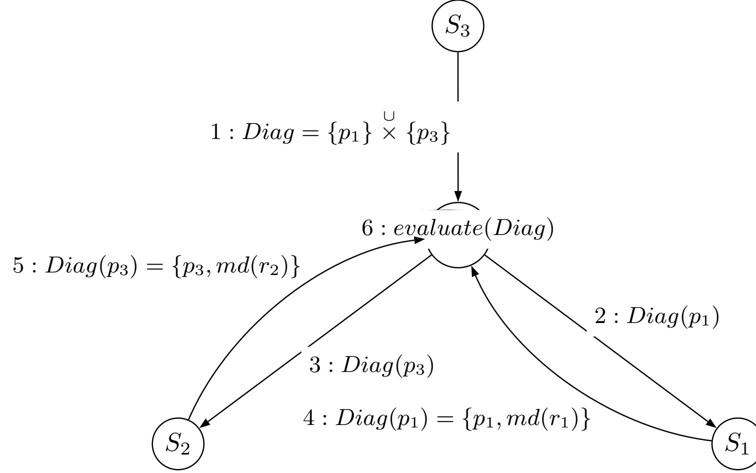


Figure 6.6: Dining philosophers: decentralized diagnosis process

The diagnosis process (see figure 6.6) starts on S_3 where t_3 detect a symptom on place p_{13} and p_{33}. So the local diagnoser $\mathcal{LD}_3$ on S_3 constructs an inequations system (algorithm 6.1) by applying the incidence equation (algorithm 6.1). By performing the local diagnosis algorithm 5 LDS, the primary local diagnosis D_0 is $\{p_1\} \overset{\cup}{\times} \{p_3\}$ in which $RP^{in}_{1\to 3} = \{p_1\}$ and $RP^{in}_{2\to 3} = \{p_3\}$. D_0 is then sent to the $inQueue$ of the coordinator for further diagnosis (algorithm 7 GDS). The coordinator pope this request from its $inQueue$ and separates D_0 as $\{p_1\}$ and $\{p_3\}$ then send the diagnosis requests $\langle p_1, \mathcal{LD}_1\rangle$ and $\langle p_3, \mathcal{LD}_2\rangle$ to the corresponding partners $\mathcal{LD}_1$ and $\mathcal{LD}_2$ (algorithm 7 GDS line 3).

So on $\mathcal{LD}_1$ and $\mathcal{LD}_2$, two inequations systems are constructed as illustrated in tables 6.2 and 6.3 by applying the incidence equations illustrated in tables 6.2 and 6.3. Then the coordinator receives the Local diagnosis $D_1 = \{p_1, md(r_1)\}$ and $D_2 = \{p_3, md(r_2)\}$ from $\mathcal{LD}_1$ and $\mathcal{LD}_2$ (algorithm 7 GDS line 6).

To calculate the minimal diagnosis, the coordinator applies the function $Evaluate$. The minimal diagnosis is show in the equation 4.7 in chapter 4, which is the same diagnosis result as using the centralized diagnosis algorithm.

$$\begin{cases} \mathrm{p}_1 : 0 \not\succcurlyeq * - 1`\chi_{p_1} \\ \mathrm{p}_3 : 0 \not\succcurlyeq * - 1`\chi_{p_3} \\ \mathrm{p}_{13} : r \succcurlyeq 0 + 1`FW(\chi_{p_1}) \\ \mathrm{p}_{33} : r \succcurlyeq 0 + 1`FW(\chi_{p_3}) \end{cases} \tag{6.1}$$

$$\begin{vmatrix} p_1: & * \\ p_3: & * \\ p_{13} & r \\ p_{33} & r \end{vmatrix} \succcurlyeq \begin{vmatrix} p_1: & * \\ p_3: & * \\ p_{13} & 0 \\ p_{33} & 0 \end{vmatrix} +$$

C	t3	r3
p_1	$-\chi_{p_1}$	$FW(\chi_{p_{13}})$
p_3	$-\chi_{p_3}$	$FW(\chi_{p_{33}})$
p_{13}	$FW(\chi_{p_1})$	$-\chi_{p_{13}}$
p_{33}	$FW(\chi_{p_3})$	$-\chi_{p_{33}}$

$$\times \begin{vmatrix} t_3: & 1 \\ r_3: & 0 \end{vmatrix}$$

Table 6.1: Dining philosopher: inequations system in form of matrix calculation on S_3

$$\begin{vmatrix} p_1: & r \\ p_2: & * \\ p_{11} & 0 \\ p_{21} & 0 \end{vmatrix} \succcurlyeq \begin{vmatrix} p_1: & * \\ p_2: & * \\ p_{11} & 0 \\ p_{21} & 0 \end{vmatrix} +$$

C	t1	r1	
		OK	KO
p_1	$-\chi_{p_1}$	$FW(\chi_{p_{11}})$	r
p_2	$-\chi_{p_2}$	$FW(\chi_{p_{21}})$	r
p_{11}	$FW(\chi_{p_1})$	$-\chi_{p_{11}}$	$-\chi_{p_{11}}$
p_{21}	$FW(\chi_{p_2})$	$-\chi_{p_{21}}$	$-\chi_{p_{21}}$

$$\times \begin{vmatrix} t_1: & 1 \\ r_1.OK: & n_1 \\ r_1.KO: & n_2 \end{vmatrix}$$

Table 6.2: Dining philosopher: inequations system in form of matrix calculation on S_1

$$\begin{cases} n_1 + n_2 = 1 \\ p_1 : r \succcurlyeq * - 1\text{`}\chi_{p_1} + n_1\text{`}\chi_{11} + n_2\text{`}r \\ p_2 : * \succcurlyeq * - 1\text{`}\chi_{p_2} + n_1\text{`}FW(\chi_{p_{21}}) + n_2\text{`}r \\ p_{11} : 0 \succcurlyeq 0 + 1\text{`}FW(\chi_{p_1}) - n_1\text{`}\chi_{p_{11}} - n_2\text{`}\chi_{p_{11}} \\ p_{21} : 0 \succcurlyeq 0 + 1\text{`}FW(\chi_{p_2}) - n_1\text{`}\chi_{p_{21}} - n_2\text{`}\chi_{p_{21}} \end{cases} \tag{6.2}$$

$$\begin{vmatrix} p_2: & * \\ p_3: & r \\ p_{22} & 0 \\ p_{32} & 0 \end{vmatrix} \succcurlyeq \begin{vmatrix} p_2: & * \\ p_3: & * \\ p_{22} & 0 \\ p_{32} & 0 \end{vmatrix} +$$

C	t2	r2	
		OK	KO
p_2	$-\chi_{p_2}$	$FW(\chi_{p_{22}})$	r
p_3	$-\chi_{p_3}$	$FW(\chi_{p_{32}})$	r
p_{22}	$FW(\chi_{p_2})$	$-\chi_{p_{22}}$	$-\chi_{p_{22}}$
p_{32}	$FW(\chi_{p_3})$	$-\chi_{p_{32}}$	$-\chi_{p_{32}}$

$$\times \begin{vmatrix} t_2: & 1 \\ r_2.OK: & n_3 \\ r_2.KO: & n_4 \end{vmatrix}$$

Table 6.3: Dining philosopher: inequations system in form of matrix calculation on S_2

$$\begin{cases} n_3 + n_4 = 1 \\ p_2 : * \succcurlyeq * - 1\text{`}\chi_{p_2} + n_3\text{`}FW(\chi_{p_{22}}) + n_4\text{`}r \\ p_3 : r \succcurlyeq * - 1\text{`}\chi_{p_3} + n_3\text{`}FW(\chi_{p_{32}}) + n_4\text{`}r \\ p_{22} : 0 \succcurlyeq 0 + 1\text{`}FW(\chi_{p_2}) - n_3\text{`}\chi_{p_{22}} - n_4\text{`}\chi_{p_{22}} \\ p_{32} : 0 \succcurlyeq 0 + 1\text{`}FW(\chi_{p_3}) - n_3\text{`}\chi_{p_{32}} - n_4\text{`}\chi_{p_{32}} \end{cases} \tag{6.3}$$

6.5 Proof of global consistency of decentralized diagnosis

The proof equivalence of the decentralized and centralized diagnosis is inspired by [149], in which the definition of functional Petri net and contact places is introduced. We extend these definitions into our CPN fault model.

6.5.1 Functional CPN definition

Definition 74 (Functional CPN)**.** *A* functional CPN *is a triple* $Z = \langle N, RP^{in}, RP^{out} \rangle$*, where* N *is a CPN graph,* $RP^{in} \subseteq P$ *is a set of bordered input places,* $RP^{out} \subseteq P$ *is a set of bordered output places, at that sets of input and output places do not intersect:* $RP^{in} \cap RP^{out} = \emptyset$*, and places of set* $Q = P \setminus (RP^{in} \cup RP^{out})$ *will be named an internal and places* $\mathbb{C} = RP^{in} \cup RP^{out}$ *as a contact set.*

Definition 75 (Subnet of CPN)**.** *A CPN* $N' = \langle \Sigma', \Gamma', P', T', Cd', Pre', Post', F' \rangle$ *is a* subnet *of CPN* N *if* $\Sigma' \subseteq \Sigma$, $\Gamma' \subseteq \Gamma$, $P' \subseteq$, $T' \subseteq T$, $Cd' \subseteq Cd$, $Pre' \subseteq Pre$, $Post' \subseteq Post$, $F' \subseteq F$.

Functional net $Z = \langle N', RP^{in}, RP^{out} \rangle$ is named as *functional subnet* of CPN N and denoted as $Z \succ N$ if CPN N' is a subnet of CPN N.

Definition 76 (Minimal subnet of CPN)**.** *Functional subnet* $Z' \succ N$ *is minimal if it does not contain any other functional subnet of CPN of the source CPN* N.

In global view of the decentralized diagnosis architecture, the complete CPN for diagnosis is naturally composed of functional subnet of CPN, and the contact set is the union of the input/output bordered places set. And we can now enumerate the most significant properties of functional subnets of CPN:

1. Functional CPN subnet is generated by the set of its own transitions.
2. Set of minimal functional CPN subnets $\mathfrak{Z} = \{Z^j\}$, $Z^j \succ N$ defines the partition of set T into nonintersecting subsets T^j, s.t. $T = \bigcup_j T^j$, $T^j \cap T^k = \emptyset$, $j \neq k$.
3. Each functional CPN subnet Z' of an arbitrary CPN N is the sum (union) of finite number of minimal functional CPN subnets. Union of subnets may be defined with the aid of operation of contact places fusion.

4. Each contact place of decomposed CPN has no more than one input minimal functional CPN subnet and no more than one output minimal functional CPN subnet (the corresponding property of PN is proved in [148]).

5. For a bordered places set $RP^{in}_{j\rightarrow i} = RP^{out}_{j\rightarrow i} \subseteq \mathbb{C}$ in a distributed environment, $\forall p \in RP^{in}_{j\rightarrow i}$, $M_{0i}(p) = M_{0j}(p) + C(p,\cdot) \times \overrightarrow{\delta}_j$ which means the initial color of p in the functional subnet Z^i is decided by the firing of functional subnet Z^i.

6.5.2 Fundamental equations of functional subnets

Consider an inequations system:

$$\hat{M} \succcurlyeq M_0 + C \times \overrightarrow{\delta} \tag{6.4}$$

Each equation $Eq_{p_i} : \hat{M}(p_i) \succcurlyeq M_0(p_i) + C(p_i, .)\overrightarrow{\delta}$ where $C(p_i, .)$ represents the row of p_i in the incidence matrix C. Therefore the system 6.4 may be represented as:

$$\text{Eq=Eq}_{p_1} \wedge Eq_{p_2} \wedge \cdots \wedge Eq_{p_n}. \tag{6.5}$$

Theorem 1. *Solution $\overrightarrow{\delta'}$ of inequations system Eq (see equation 6.5) for CPN N is the solution of inequations system for each of its functional CPN subnets Eq_{p_i}.*

Proof. As $\overrightarrow{\delta'}$ is the solution of inequations system for CPN N , so $\overrightarrow{\delta'}$ is a CPN diagnosis solution of system 6.5 and consequently $\overrightarrow{\delta'}$ is a CPN diagnosis solution for each of equations Eq_{p_i}. Thus $\overrightarrow{\delta'}$ is a solution for an arbitrary subset Eq_{p_i}. According to property 1, a functional CPN subnet Z' , $Z' \succ N$ is generated by the set of its own transitions T'. Thus, an inequation corresponds to a transition of subnet has the same form Eq_{p_i} as for the entire net, so subnet contains all the incident places of source net.

Therefore the inequations system for functional subnet Z' , $Z' \succ N$ is a subset of set $\{Eq_{p_i}\}$ and vector $\overrightarrow{\delta'}$ is its solution. Consequently $\overrightarrow{\delta'}$ is the solution of inequations system for functional subnet Z'. Arbitrary choice of subnet $Z \succ N$ in above reasoning proves the theorem. □

Theorem 2. *Inequations system of Petri net is solvable if and only if it is solvable for each minimal functional subnet and a common solution for contact places exists.*

Proof. According to property 2, a set of minimal functional subnets $\mathfrak{Z} = \{Z^j\}$, $Z^j \succ N$ of an arbitrary CPN N defines a partition of set T into nonintersecting subsets T_j. Let number of minimal

functional CPN subnets equals k. As mentioned in the proof of theorem 1, the equations contain the terms for all the incident places. Therefore,

$$L \Leftrightarrow L^1 \wedge L^2 \wedge \cdots \wedge L^k. \tag{6.6}$$

where L^j is a subsystem for a minimal functional CPN subnet Z^j, $Z^j \succ N$.

To solve the inequations system $Q_{\hat{M}}$ which are composed by L^j, two conditions should be satisfied as follows:

(i) for the corresponding equations $|L^j|$ which considers the token number, the left and right sides of each equation are equal;

(ii) for the inequations system $Q_{\hat{M}}$, the covering relations ($\succcurlyeq$) must hold for each inequation.

So the proof of the correctness of the distributed diagnosis, the two conditions should be both proved:

(i) The number of token of multi-set for CPN satisfy the property 2, too. So the equation 6.7 holds.

$$|L| \Leftrightarrow |L^1| \wedge |L^2| \wedge \cdots \wedge |L^k|. \tag{6.7}$$

where $|L^j|$ represents the token number of the corresponding functional subnet L^j.

Note that if $|L^j|$ has not solutions, than $|L|$ has not solutions, too. Let a general solution for each functional CPN subnet has the form

$$\bar{x}^j = \bar{x}'^j + u^j \cdot G^j \tag{6.8}$$

where $u^j \cdot G^j$ is the general solution of homogeneous system, $\bar{x}'^j \in X'^j$, where X'^j is the set of minimal particular solution of nonhomogeneous system of equations.

According to equations 6.7 and 6.8:

$$|L| \Leftrightarrow \bar{x}'^1 + u^1 \cdot G^1 = \bar{x}'^2 + u^2 \cdot G^2 = \cdots = \bar{x}'^k + u^k \cdot G^k \tag{6.9}$$

Therefore system

$$\bar{x} = \bar{x}'^1 + u^1 \cdot G^1 = \bar{x}'^2 + u^2 \cdot G^2 = \cdots = \bar{x}'^k + u^k \cdot G^k \tag{6.10}$$

is equivalent to source system

$$|\hat{M}| = |M_0| + |C| \times \overrightarrow{\delta} \tag{6.11}$$

We need to demonstrate the solution of system 6.10 requires smaller quantity of equations. Consider a set of places of CPN N with the set of minimal functional subnets $\{Z^j | Z^j \succ N\}$:

$$P = Q^1 \cup Q^2 \cup \cdots \cup Q^k \cup \mathbb{C} \tag{6.12}$$

where Q^j is a set of internal places of subnet Z^j and $\mathbb{C}$ is a set of contact places. According to definition each internal place $p \in Q^j$ is incident only to transitions from set T^j. Thus x_p corresponding to this places is contained only in system L^j. Consequently, it is only necessary to solve the equations for contact places from set $\mathbb{C}$.

Now construct equations for the contact places of net $p \in \mathbb{C}$, so they are only incident more than one subnet. According to property 4, each contact place $p \in \mathbb{C}$ is incident not more than two functional subnets. Therefore, we get equation:

$$\bar{x}_p'^j + u^j \cdot G_p^j = \bar{x}_p'^l + u^l \cdot G_l^j \tag{6.13}$$

where j, l is the numbers of minimal functional subnets incident to contact place $p \in \mathbb{C}$ and G_p^j is a column of matrix G^j corresponding to place p. Equation 6.13 may be transformed as:

$$u^j \cdot G_p^j - u^l \cdot G_l^j = \bar{x}_p'^l - \bar{x}_p'^j \tag{6.14}$$

Thus system

$$\begin{cases} x_p = \bar{x}_p'^j + u^j \cdot G_p^j, p \in Q^j \vee p \in \mathbb{C}, \\ u^j \cdot G_p^j - u^l \cdot G_l^j = \bar{x}_p'^l - \bar{x}_p'^j, p \in \mathbb{C} \end{cases} \tag{6.15}$$

is equivalent to the source system 6.11.

(ii) Given two communicating functional subnets Z^j and Z^l, which communicate through $rp_{j\to l}^{out}$, which means subnet Z^j sends a message $\mathbb{C}_{j\to l} = rp_{j\to l}^{out}$ to Z^l. The diagnosis results of these two functional subnets are respectively $Diag^j$ and $Diag^l$. To get the union of the diagnosis of two communicating functional subnet, a communicating functional subnet union operator is defined as follows:

Definition 77 (Communicating functional subnet union). $\underset{\mathbb{C}}{\overset{\cup}{\times}}$ *is a communicating functional subnet union operator which define as follows:*

(i) if $\mathbb{C}^l_{j\to l} = \emptyset$, $Diag^l \underset{\mathbb{C}}{\overset{\cup}{\times}} Diag^j = Diag^l \cup Diag^j$,

(ii) if $\mathbb{C}^l_{j\to l} \neq \emptyset$, $Diag^l \underset{\mathbb{C}}{\overset{\cup}{\times}} Diag^j = Diag^l_{\hat{M}^l \setminus \hat{M}(\mathbb{C}_{j\to l})} \overset{\cup}{\times} Diag^{j,l}_{\hat{M}(\mathbb{C}_{j\to l})} \overset{\cup}{\times} Diag^j_{\hat{M}(\mathbb{C}_{j\to l})}$

with $\hat{M}^j$ and $\hat{M}^l$ represent the symptom markings of functional subnet Z^j and Z^l. $\hat{M}(\mathbb{C}_{j\to l})$ represents the symptom marking which can inferred to the set of $\mathbb{C}_{j\to l}$. $Diag^{j,l}$ represents the diagnosis of the union of functional subnets Z^j and Z^l. $Diag^{j,l}_{\hat{M}(\mathbb{C}_{j\to l})}$ represent the diagnosis of union of the functional subnets Z^j and Z^l concerns the symptom markings $\hat{M}(\mathbb{C}_{j\to l})$.

According to the decentralized diagnosis algorithm, the global diagnosis is calculated as $Diag = \sum_{\mathbb{C}^l_{j\to l} \neq \emptyset} Diag^l \underset{\mathbb{C}}{\overset{\cup}{\times}} Diag^j$[1]. The principle of the diagnosis processes is to start from the symptom and compute the diagnosis $Diag^l$, and continue to infer according to the bordered communication places $\mathbb{C}^l_{j\to l}$, and corresponding expressions concerns a bordered places p in $Diag^j$ is $-\chi_p$, a negative item which is ignored during the diagnosis process. So the global inequations system can be separated as the local inequations systems. Each local inequations system includes two parts: a set of inequations of a bordered place, which are in form of $0 \succcurlyeq * - 1`\chi_p$ on the local diagnoser l (which leads to $\emptyset$ during the diagnosis process); and a set of inequations concern the internal places on the local diagnoser j. Then the diagnosis concerns the bordered places computed by the local diagnoser j is equivalent to that of by the global diagnoser. Thus the communicating functional subnet union of all the local diagnosis is equal to the global diagnosis.

The equivalence of the global counter-diagnosis with the union of the local counter-diagnosis can be proved in the same way.

□

6.6 Decentralized diagnosis of orchestrated BPEL services

In order to enhance fault management in complex services with the capability of reasoning on global failures of the overall service, we propose to:

- Associate with each basic service a local diagnoser to provide the coordinator with the information needed for identifying causes of an exception.

[1] The local diagnosis are based on the updated local inequations system according to the counter-diagnosis

- Provide a coordinator, which is not tied to any specific service, but is able to invoke local diagnosers and relate the information they provide, in order to reach a diagnosis for the overall complex service. In case the supply chain has several levels, several coordinators may form a hierarchy, where a higher-level coordinator sees the lower level ones as local diagnosers.

Each local diagnoser interacts with its Web Service and with the coordinator. The coordinator interacts only with local diagnosers. More precisely, the interaction follows this pattern:

- During service execution, each local diagnoser should monitor the activities carried out by its Web Service, logging the messages it exchanges with the other peers. The diagnoser exploits an internal ”observer” component collecting the messages and locally saving them for later inspection. Notice that when a Web Service composes a set of sub-suppliers, the coordinator of the sub network of cooperating services must fill the local diagnoser role. On the other hand, a Web Service can have a basic local diagnoser that does not need to exploit other lower-level diagnosers in order to do its job. Local diagnosers need to exploit a model of the Web Service in their care, and is able to construct an inequations system to perform local diagnosis task.
- When a local diagnoser receives an alarm message, it starts reasoning about the problem to identify its possible causes, which may be internal to the Web Service or external (erroneous inputs from other services). The diagnoser can do this by analyzing the messages it previously logged.
- The local diagnoser informs the coordinator about the alarm it received and the hypotheses it made on the causes of the error. The coordinator starts invoking other local diagnosers (following a diagnostic reasoning pattern, detailed in section 6.4.1) and relating the different answers, in order to reach one or more global candidate diagnoses that are consistent with reasoning performed by local diagnosers.

From the communication point of view, the inclusion of local and global diagnosers in the architecture of a complex Web Service is relatively seamless because diagnosers can be implemented as Web Services (local/coordinator WS) interacting with the other peers via WSDL messages. Specifically:

- Local diagnosers must offer a WSDL operation (logMessage (String wsdlMsg)) for the reception of the messages to be logged.

- Each Web Service must send copies of the inbound and outbound messages to its local diagnoser. To this purpose, each Web Service must be equipped with a ”logging service” proxy which interceptWSDL messages and sends a copy of each message to Local Diagnoser WS through the ”logMessage” port.
- The coordinator must offer a WSDL operation to be used by local diagnosers to trigger the global diagnostic process.
- Local diagnosers must offer a WSDL operation to be used by the coordinator in order to invoke them.

The proposed approach is modular and supports a seamless introduction of advanced fault reasoning in the management of complex Web Services. The key point is that local diagnosers can exploit specialized reasoning techniques without imposing the same techniques on any of the involved Web Services. Although we require that Web Services notify local diagnosers about (normal and fault) messages they receive from or send to other services, this feature can be added to the invoked services without changing their internal structure. Moreover, if one of the involved services does not have a local diagnoser, or the model of the service exploited by the local diagnoser is very rough, the coordinator can still perform its job but the results may be less precise (e.g., it may not be possible to rule out the non-diagnosed service as the cause for the error).

6.7 Case study:foodshop

6.7.1 Exceptions

Theoretically, four classes of exceptions are possible during the whole foodshop process.

CUSTomer exceptions:

- WrongBillException. CUST checks the bill and realizes that there is something wrong (missing and/or unwanted items) (just before $replyPay$ activity in figure 5.7)
- TimeOutException. CUST is waiting for some feedback from the shop (either an unavailability notification, or a request for payment) but none of the two takes place (just before $replyPay$ activity in figure 5.7).

- WrongParcelException. CUST receives a parcel with missing and/or unwanted items (upon $receive$ in figure 5.7).

- TimeOutException. CUST never receives the parcel (just before $receive$ in figure 5.7).

SHOP exceptions:

- WrongAnswerException. For some items the answer from WAREHOUSE/SUPPLIER is missing, or the answer is about a different item than asked for (upon $ReplyAllIsAvailable$ in figure 5.6).

- TimeOutException. The SHOP never receives an answer on item availability either from the WAREHOUSE or from the SUPPLIERs. (in the middle of $Supplier : CheckAndReserve$ and $Warehouse : CheckAndReserve$ in figure 5.6 which are in fact respectively consist of an invoke and a receive activity).

- HighShipCostException. The shipping cost sent from the WAREHOUSE is higher than an expected threshold.

- TimeOutException. The SHOP never receives an answer on the ship cost from the WAREHOUSE (just before $SendCustomerPackage$ in Warehouse process in the figure 5.6).

- TimeOutException. The SHOP never receives an answer from the CUSTomer on whether he/she wants to pay or not (just before Paid in figure 5.7).

  SUPPLIER exceptions:

  - WrongResCodeException. The reservation code is not recognized by SUPPLIER (either upon the activity $CancelOrder$ or upon $ConfirmOrder$ of supplier in figure 5.6).
  - TimeOutException. The buyer (SHOP or WAREHOUSE) never tells SUPPLIER whether to cancel the order or proceed with it (after $CancelOrder$ of shop or warehouse in figure 5.6).

  WAREHOUSE exceptions:

  - TimeOutException. Some answers on item availability never arrive from the SUPPLIERs. (just before $ReturnUnavailableList$ of warehouse in figure 5.6).

- WrongAnswerException. For some items the answer from the SUPPLIERs is missing, or the answer is about a different item than asked for (upon $AssembleShipment$ of warehouse in figure 5.6).
- TimeOutException. The WAREHOUSE never receives from the SHOP an answer on whether to cancel the reservation or to proceed computing the ship cost (after $ReturnUnavailableList$ of warehouse in figure 5.6).
- TimeOutException. After providing the ship cost, the WAREHOUSE never receives an answer from the SHOP on whether to cancel or to proceed with the order (after $ReturnUnavailableList$ of warehouse in figure 5.6).
- WrongSupplyException. Some items that arrive from the suppliers are wrong (upon assemble).
- TimeOutException. Some items never arrive from the SUPPLIERs (upon $AssembleShipment$ of warehouse in figure 5.6).

6.7.2 Fault scenarios

In this section we highlight some failure situations within the process. In the following section, we will describe a sample diagnostic process for each of these situations.

We will study three situations that are started by an exception:

1. When computing the bill (activity $CalculateTotalCost$), the SHOP realizes that the ship cost sent by the WAREHOUSE is higher than the expected threshold ($HighShipCostException$ of the SHOP).
2. When receiving the bill, the CUSTomer realizes that some ordered item is wrong ($WrongBillException$ of the CUSTomer).
3. When assembling the package, the WAREHOUSE realizes that it received (activity) a wrong item from one of the SUPPLIERs ($WrongSupplyException$ of the WAREHOUSE).

From the point of view of diagnosis, exceptions are symptoms of faults. There can be several possible causes for an exception; diagnosis must discard those that cannot have happened (based on observations), possibly reducing the possibilities to the one that took place.

1. There can be two causes for a $HighShipCostException$ in the shop: either the SHOP selected the wrong warehouse (thus choosing one that is far from the customer address), or

the warehouse itself made a mistake (activity $PrepareCustomerPackage$) in computing the ship cost. Diagnostic reasoning can find these two possible causes with backward reasoning, but without adding any observable data or test action it is not possible to discriminate between the two.

2. A $WrongBillException$ is caused by someone reserving the wrong item, either the WAREHOUSE (activity $Availability$)) or one of the SUPPLIERs (activity $Availability$). By following backwards the path of the wrong item data, it is possible to discover who reserved that particular item and correctly diagnose the problem.

3. Let us look at the possible causes for a $WrongSupplyException$. Apparently there are three possibilities:

   (a) the SUPPLIER reserved (activity $Availability$) the wrong item from the beginning;

   (b) the SUPPLIER reserved the correct item but then made a mistake in updating its internal order DB, writing the wrong item code (behind the activity $Availability$) ;

   (c) the SUPPLIER did everything correct but sent (activity $SnedShipment$) the wrong parcel to the WAREHOUSE.

   However, possibility 3a can be discarded by observing that it would have produced an error in the bill, if the bill is asserted as correct. Thus only possibilities 3b and 3c remain as candidates. The SUPPLIER could discover the source of the error by comparing the reservation codes it sent to the SHOP with those it wrote down in its DB: if they are the same then 3c holds; otherwise 3b holds. It is worth noting that this further check is not included in our CPN model, so we cannot distinguish the faults 3b and 3c.

Assume a customer selects a food ware from the on-line site, and that his request includes the phases of food checking for availability (service activated by the customer), food selection from a warehouse (service activated by the shop service), book shipping (service executed by the shipper service), and payment (service by an external payment service). Suppose that the Shipper, Warehouse, and Supplier belong to a trust circle, that is, that no security faults can occur in the messages exchanged among these three services. Faults that may arise in the trust circle are a resource-booking fault, due to mismatch of resource reservation to execute the application. An internal data fault may occur when the Shop sends order data to the Warehouse (e.g., a wrong ID).

Another fault, of type Unavailable goods may occur during the execution of the Warehouse service, needing to store a log that asks to postpone the goods search process until a new event (Good

in Stock) arises to signal that the Warehouse has been refilled. If we view the whole application as a workflow composed of three phases: Selection-and-Booking, Payment, and Delivery, a fault of type phase time out occurs if one phase exceeds the foreseen time schedule; a session fault occurs if a connection is lost among the phases, and the collected data are lost.

Finally, consider that food reservation, payment, and shipping are regarded as services that have been orchestrated and attached to the customer context through e.g., the customer's mobile device or browser. If the shipping service arises a fault, e.g., a missed delivery due to a delayed delivery time, we regard this fault as a QoS violation in terms of delivery service time fault, which is not discussed in this thesis.

An important aspect that has to be considered in this example is the presence of many actors, each with its own database. Since the partners are involved in the same business, databases could overlap and thus be affected by data misalignments. There could be database misalignments between shop and Warehouse and consequently the shop has out of date catalogues. In some cases this fault might imply the mismatch of customers' requirements. In fact, it could happen that a warehouse does not communicate to the shop price variations. In this case, shop, along the customers' requirements and the available information contained in its own databases, might select that warehouse but the new prices do not satisfy the request. Customer would receive a bill higher than the requested one. The error is due to the low values of timeliness associated with data owned by the shop. Misalignments between shop and other actors' databases can also cause completeness problem. The partners need therefore to analyze their communication processes and adopt efficient synchronization mechanisms by choosing the most suitable time interval to perform the periodic realignment among databases.

The Diagnosis on WRONG PRICE determines the possible wrong components and actions, such as:

- Wrong computation by SHOP
- Wrong ship cost by WAREHOUSE
- Wrong data in catalogue by SHOP/WAREHOUSE (low data quality)
- Wrong formulation of problem by CUSTOMER
- Wrong communication (dialog) e.g. 20$ or 20

Considering as a final example the fault: *INCORRECT ASSEMBLE OF PARCEL*, we have the following schema:

Diagnosis: before assemble of parcel action Possible wrong components:

- Wrong synchronization by SUPPLIER (e.g., goods are reserved but not available)
- Wrong parts by SUPPLIER
- Wrong reservation by SUPPLIER
- Missing parts by SUPPLIER
- Wrong parcel composition by WAREHOUSE

6.7.3 Diagnosis

This section applies the decentralized diagnosis approach for the fault *INCORRECT ASSEMBLE OF PARCEL*. The activity names of the foodshop are abbreviated as in tables 6.4- 6.7. Table 6.8 lists the communicating messages (a band of bordered places) between the partners CUSTomer, SHOP, SUPPLIER, WAREHOUSE, and LocalSupplier.

ReceiveOrder	t_1	StoreOrder	t_2	SplitOrder	t_3
Supplier:: CheckAndReserve	t_4	Warehouse:: CheckAndReserve	t_5	CalculateTotalCost	t_6
ReplyAllIsAvailable	t_7	ReceiveConfirmation FromCustomer	t_8	ForwardOrder	t_9
Supplier:: ConfirmOrder	t_{10}	Warehouse:: ForwardOrder	t_{11}	CancelOrder	t_{12}
Supplier:: CancelOrder	t_{13}	Warehouse:: CancelOrder	t_{14}	ReplyCustomer Confirmation	t_{15}
MergeUnavailableList	t_{16}	ReplySomething IsNotAvailable	t_{17}		

Table 6.4: The activity names abbreviation of $Shop$ service

CheckAndReserve	t_{18}	CancelOrder	t_{19}	Availability	t_{20}
Return UnavailableList	t_{21}	ConfirmOrder	t_{22}	AssembleShipment	t_{23}
SendShipment	t_{24}				

Table 6.5: The activity names abbreviation of $RealSupplier$ service

CheckAndReserve	t_{25}	Availability	t_{26}	ReserveOn LocalSupplier	t_{27}
ReceiveLocalSupplier Shipment	t_{28}	Return UnavailableList	t_{29}	CancelOrder	t_{30}
ForwardOrder	t_{31}	ReceiveSupplier Shipment	t_{32}	PrepareCustomer Package	t_{33}
SendCustomer Package	t_{34}				

Table 6.6: The activity names abbreviation of $Warehouse$ service

ReceiveOrderFrom Wharehouse	t_{35}	PrepareLocalSupplier Shipment	t_{36}	SendLocalSupplier Shipment	t_{37}

Table 6.7: The activity names abbreviation of $LocalSupplier$ service

Source		Message	Target	
Service	Activity		Activity	Service
CUSTomer	unknown	$ShopWSSEI_receiveOrder$	t_1	SHOP
SHOP	t_4	$SupplierWSSEI_checkAvailReserve$	t_{18}	SUPPLIER
SUPPLIER	t_{21}	$RealSupplierAnswerMSG$	t_4	SHOP
SHOP	t_5	$WarehouseWSSEI_checkAvailable$	t_{25}	WAREHOUSE
WAREHOUSE	t_{29}	$WHAnswerMSG$	t_5	SHOP
SHOP	t_{10}	$ConfirmMSG$	t_{22}	SUPPLIER
SHOP	t_{11}	$WarehouseWSSEI_ForwardOrder$	t_{31}	WAREHOUSE
SHOP	t_{13}	$SupplierWSSEI_unReserve$	t_{19}	SUPPLIER
SHOP	t_{14}	$WarehouseWSSEI_unReserve$	t_{30}	WAREHOUSE
SUPPLIER	t_{24}	$SupplierWSSEI_ShippingRequest$	t_{32}	WAREHOUSE
WAREHOUSE	t_{27}	$ReservationRequest$	t_{35}	LocalSupplier
LocalSupplier	t_{37}	$ReservationResponse$	t_{28}	WAREHOUSE
SHOP	t_{17}	$ShopWSSEI_reply2Client$	unknown	CUSTomer

Table 6.8: The communication messages shared by the partners

From	to	Diag name	Diag value
Mon_3	LD_3	$Diag_0$	$Diag(p_1) \overset{\cup}{\times} Diag(p_12) \overset{\cup}{\times} Diag(p_3)$
LD_3	Cor	$Diag_3^{(1)}$	$\{md(t_{34})\}$
			$\{md(t_{33}), p_4\} \overset{\cup}{\times} \{\{p_5\} \overset{\cup}{\times} \{p_6, p_7\}\}$
Cor	LD_2		$Diag(p_4)$
Cor	LD_1		$Diag(p_5)$
Cor	LD_1		$Diag(p_6)$
Cor	LD_4		$Diag(p_7)$
LD_2	Cor	$Diag_2^{(1)}$	$Diag(p_4) = \{md(t_{24}), md(t_{23}), md(t_{20}), p_{11}\}$
LD_1	Cor	$Diag_1^{(1)}$	$Diag(p_5) = \{md(t_5), p_8\}$
LD_1	Cor	$Diag_1^{(2)}$	$Diag(p_6) = \{md(t_5), p_9\}$
LD_4	Cor	$Diag_4^{(1)}$	$Diag(p_7) = \{md(t_{36}), p_{10}\}$
Cor	LD_1		$Diag(p_{11})$
Cor	LD_1		$Diag(p_8)$
Cor	LD_1		$Diag(p_9)$
LD_1	Cor	$Diag_1^{(3)}$	$Diag(p_{11}) = \{md(t_4), p_{12}\}$
LD_1	Cor	$Diag_1^{(4)}3$	$Diag(p_8) = \{p_{13}\}$
LD_1	Cor	$Diag_1^{(5)}$	$Diag(p_9) = \{p_{14}\}$
Cor	LD_3		$Diag(p_{10})$
LD_3	Cor	$Diag_3^{(2)}$	$Diag(p_{10}) = \{md(t_{26}), md(t_9)\}$
Cor	Cor		$Diag = \{\{md(t_{34})\}, evaluate(\{md(t_{33}), p_4\} \overset{\cup}{\times} \{\{p_5\} \overset{\cup}{\times} \{p_6, p_7\}\})$

Table 6.9: The decentralized diagnosis process of foodshop example for exception INCORRECT ASSEMBLE OF PARCEL

Concerns the fault scenario INCORRECT ASSEMBLE OF PARCEL, the distributed observation can beAs there is not loop in each BPEL service, the occurrence of each activity is 1, which is omitted:

On $shop$ service (S_1), the partial order observation $(S(\mathcal{T}_1), \lhd_1)$=(($t_1$, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{15}), ((t_1,t_2), (t_2,t_3), (t_3,t_4), (t_3,t_5), (t_4,t_6), (t_5,t_6), (t_6,t_7), (t_7,t_8), (t_8,t_9), (t_9,t_{10}), (t_{10},t_{11}), (t_{11},t_{15})).

On $RealSupplier$ service (S_2), the partial order observation $(S(\mathcal{T}_2), \lhd_2)$=(($t_{18}$, t_{20}, t_{21}, t_{22}, t_{23}, t_{24}),((t_{18},t_{20}), (t_{20},t_{21}), (t_{21},t_{22}), (t_{22},t_{23}), (t_{23},t_{24})).

On $Warehouse$ service (S_3), the partial order observation $(S(\mathcal{T}_3), \lhd_3)$=(($t_{25}$, t_{26}, t_{27}, t_{28}, t_{29}, t_{30}, t_{31}, t_{32}, t_{33}, t_{34}),((t_{25},t_{26}), (t_{26},t_{27}), (t_{27},t_{28}), (t_{28},t_{29}), (t_{29},t_{30}), (t_{29},t_{31}), (t_{31},t_{32}), (t_{32},t_{33}), (t_{33},t_{34})).

On $LocalSupplier$ service (S_4), the partial order observation is in fact a complete order one:

$(S(\mathcal{T}_4), \lhd_4)$=(($t_{35}$, t_{36}, t_{37}), ((t_{35},t_{36}), (t_{36},t_{37})).

So the symptom marking is arisen by the CUSTomer of the Foodshop service and is projected on the output variable part $ArrangeShippingResponse/IncomingItems$, which is a set of bordered data places p_1 (perishable food "White bread" instead of "Brown bread"), p_2 (imperishable food "Orange juice" instead of "Lemon juice") and p_3 (imperishable food "Green tea" instead of "Indian tea") in the CPN model of the reply activity t_{34}. The local diagnoser LD_3 is triggered by this symptom marking and get the primary diagnosis $Diag_3^{(1)}$, the diagnosis stops after a series of local diagnosers invocations and coordinations (see table 6.9). Figure 6.7 illustrates the partners of the Foodshop example and the bordered date places in the circled numbers, which concerns the diagnosis.

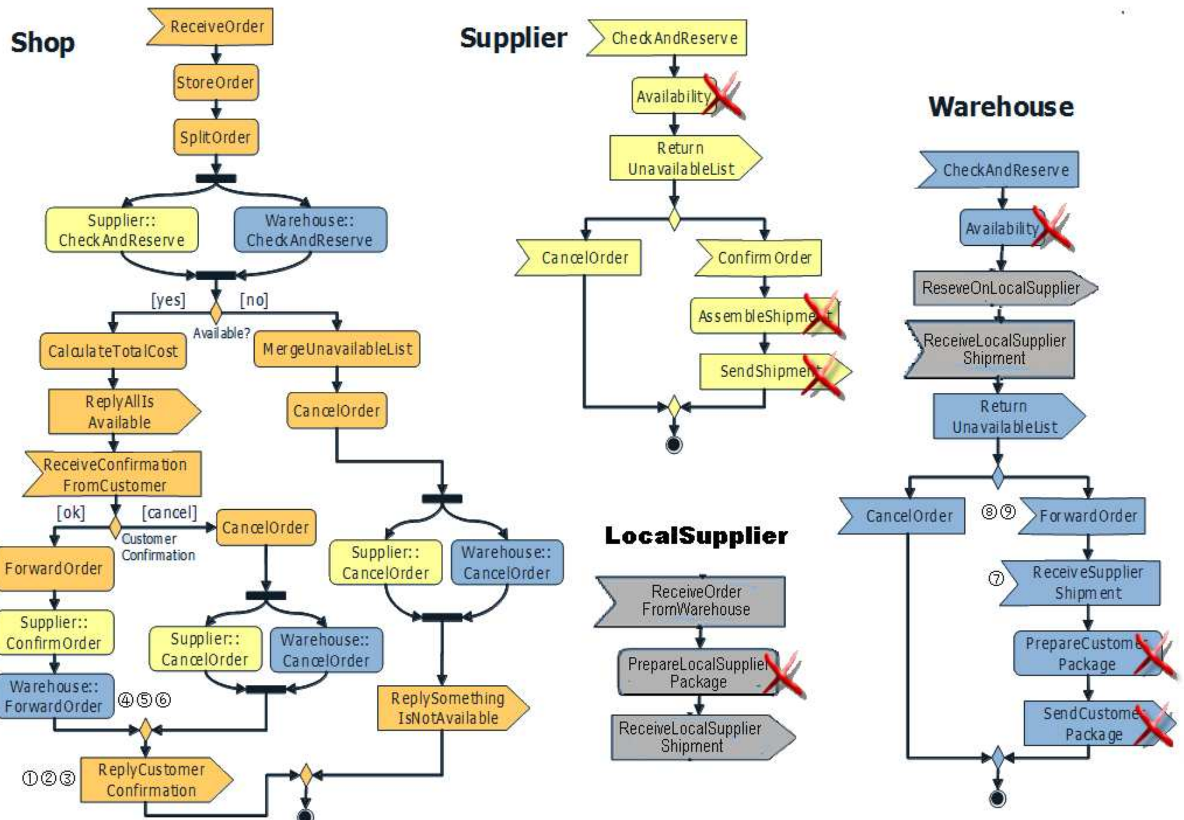


Figure 6.7: The partners of the foodshop example

Thus the diagnosis can be evaluated as $Diag = \{\{md(t_{34}\}), \{md(t_{33})\}, D_1 \overset{\cup}{\times} D_2\}$ with $D_1 = \{md(t_{24}), md(t_{23}), md(t_{20}), p_{12}\}$ and $D_2 = \{\{p_{13}\} \overset{\cup}{\times} D_3\}$ with $D_3 = \{p_{14}, md(t_{36}), md(t_{26})\}$.

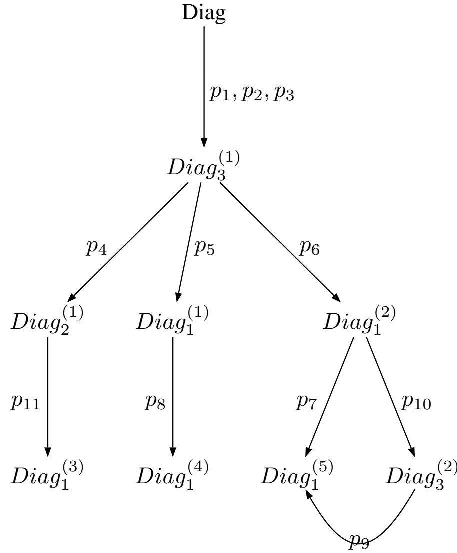


Figure 6.8: The diagnosis tree of for fault *INCORRECT ASSEMBLE OF PARCEL*: the labels on the edges represent the bordered data places need to solve further

While the input places of $Shop$ service p_{12}, p_{13}, and p_{14} are confirmed to be correct by the CUS-Tomer, they are discarded from the diagnosis. Figure 6.8 illustrates the local diagnosis computed on the local diagnosers. Then the diagnosis can be classified as (see figure 6.7):

- Wrong synchronization by SUPPLIER (e.g., goods are reserved but not available): $md(t_{23})$;
- Wrong reservation/shipment by SUPPLIER: $md(t_{20})$/$md(t_{24})$;
- Missing parts by SUPPLIER: $md(t_{20})$;
- Missing parts by Warehouse: $md(t_{26})$;
- Wrong parcel composition/shipment by WAREHOUSE: $md(t_{33})$/$md(t_{34})$;
- Wrong reservation by LocalSupplier: $md(t_{36})$;

Chapter 7

Conclusion

This research is motivated by the interest of designing distributed fault diagnosis for the models of workflows. These workflows are created according to the existing standard and languages that are used for web-services workflow modeling, and heuristic descriptions of failure situations within workflow. The workflows can be composed as a large and complex software system in which the components are located in different sites and communicate with each other by visible sent/received messages. The diagnosis problem is viewed in this thesis as part of a broader supervisory architecture taking into account that the diagnosis result is used for taking repair/reconfiguration actions to realize the ”self-healing” Web services applications. The distributed setting that we considered is very general considering that the software system comprises several components that are associated with the local monitoring components and are the local diagnosis components are coordinated by the decentralized coordinator to manage the information exchange and global diagnosis calculation.

To the best of our knowledge, it is new to model the distributed software system with the CPNs to represent both the control and data for diagnosing algebraically. In [60, 51, 86, 28, 150, 145, 77, 15], the CPNs model is used to model the Workflow from the aspects of data. The control is normally modeled as the ”guards” (the transition firing conditions) with depend much on the value of data. To diagnosis Web service based on the CPNs model is discussed in [143, 112] (CPN) and [87, 12, 4, 58, 121, 18, 41, 17, 131, 46] (PN). A distributed setting based on PN models was considered in [8, 97, 71, 97, 96]. While most of them handle the repeating processes in a meticulous way: to unfold the execution according to the observation, which is time/space consuming. Another difficulty comes from the incomplete knowledge of system status and the observation order. [113, 96, 12, 110] discussed the diagnosis of the partially observation of system states and observable events in form of Petri nets

In this thesis, a novel algebraic backward diagnosis approach is proposed to handle the repeating and partial order observation (section 4.3) more effectively without loosing the diagnosis precision. Meanwhile the incomplete system status is represented as an ”unknown” color (section 3.3.1), and the complexity of the diagnosis algorithm does not increase.

In our CPN model, the faults are of two sources: the faulty input places, or the faulty transition modes. The symptoms are represented as a symptom marking which contains the exclusive correct/faulty/unknown status of each system variable. The task of diagnosis is to explain the symptom marking by assigning to each CPN input place a correctness status and to each transition a series of modes. As a workflow process can be very complicate, sometimes choosing a wrong path can cause exceptions same as receiving faulty input data. So to unify this case, both the system data and control are modeled as the places of CPN model. The data dependency expressions between the input/output places, which are defined on the arcs of the CPNs, defines the fault transformation paths (section 3.3.1). Thus by defining the data dependencies between the control and data places, the different sorts of faults (data/control faults caused by data, control, or both) are naturally unified (section 5.5.2). In this way, the CPN model is in fact a complicate and complete cause-and-effect paths net between the input and output places.

From the point view of PN diagnosis, our task is to explain the relation between the initial and final markings, which both are a (incomplete) system variables assignments. We define a ”covering” relationship to represent the effect-and-cause relation between the symptom marking and the initial marking. And thanks to the mathematical properties of CPN, this relationship can be represent in an algebraic way: the incidence equation. So our diagnosis approach transforms the diagnosis problem as solving an inequations system. As the symptom marking contains more assured information, so the backward inferring algorithms are designed which starts from the final symptom marking and search for all reasonable assignment for the input places and transition modes (section 4.2). In case of multiple faults occur in the symptom marking, a multi fault operator $\overset{\cup}{\times}$ is defined to integrate the diagnosis results of each single fault. The pure numeric incidence equation of PN loses the order of the transitions, while in the incidence equation of the CPNs model, the partial orders are kept in the data dependency functions. Thus the algebraic approach does not violate the minimality of the diagnosis for the partial ordered observation.

This algebraic diagnosis approach can be performed either in centralized or decentralized (section 6.2) manner. In a distributed architecture, the whole system is looked as a set of place-bordered CPNs models located on different sites. The data and controls are passed by the bordered places (remote activation and data places) between the neighbor CPNs. On each site, the local diagnoser has

its own CPN models and observations. A coordinator is in charge of managing the communications between the local diagnosers through the knowledge of bordered places of the local CPN models. The backward search is used in the decentralized algorithm for deriving the preliminary local calculation of a component (Section 6.4.1). The coordinator will assemble the diagnosis results after all the local reasoning terminate. As both the CPN model and observations are locally independent, the decentralized diagnosis architecture can be easily scale up in an hieratical manner.

In section 6.5, the equivalence of the decentralized diagnosis and the global one is proved. In this section, the definition of ”functional CPN subnet” is introduced to represent the place-bordered CPNs. The equivalence between the ”functional CPN subnets” and the global CPNs is proved both on the aspect of the token number and token color reasoning.

In the part of the application (chapter 5), we studied the extendable XML-based workflow description language WS-BPEL and translated all its basic activities (message communication, synchronous/asynchronous remote WS invocation, etc) and structures operators (choice, loop, concurrency, etc) into our CPN model. The most subtle data dependencies between the places are retrieved by the XPath parsing.

7.1 Future work

We plan to further extend the results of this thesis in the following ways:

1. to study the diagnosis protocol for the distributed architecture that each local diagnoser recognize its own neighbors and updates its diagnosis according to the diagnosis requests of its neighborhoods.

2. to study the problem of diagnosability of the CPNs model by checking the deterministic properties of the columns of the CPNs incidence matrix. If in two transitions modes are not determinative in the incidence matrix, these two transitions cannot be determinately diagnosed.

3. to extend the fault prediction based on the data fault diagnosis. Once the faulty input data is confirmed, the prediction for the following places that are not reported to be faulty is reliable. The approach performs the forward reasoning on the CPNs model.

4. to introduce the time conception into the CPNs model by defining the ”guard” on the transitions. The time stamps of the monitoring log can be used to improve the quality of diagnosis but the clock synchronization between the different local sites need to be considered.

5. to include probabilistic information to order diagnosis result. The probabilistic information can be calculated according to the long term QoS data or the qualifications of the components providers.

Bibliography

[1] *Fault Diagnosis: Models, Artificial Intelligence, Applications*. Springer, 1 edition, February 2004.

[2] Inc. Active Endpoints. Active endpoints. `http://www.activevos.com`.

[3] M. Alcaraz-Mejia, E.Lopez-Mellado, A.Ramirez-Trevino, and I.Rivera-Rangel. Petri net based fault diagnosis of des. In *Proc. IEEE-. SMC*, pages 4730–4735, Washington, USA, 2003.

[4] Cosimo Anglano and Luigi Portinale. B-W analysis: a backward reachability analysis for diagnostic problem solving suitable to parallel implementation. In Valette, R., editor, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, pages 39–58. Springer-Verlag, 1994.

[5] Liliana Ardissono, Luca Console, Anna Goy, Giovanna Petrone, Claudia Picardi, Marino Segnan, and Daniele Theseider Dupré. Enhancing web services with diagnostic capabilities. In *European Conference on Web Services*, pages 182–191, Växjö, Sweden, 2005. IEEE CS.

[6] Salomaa Arto and Sneddon Ian N. *Theory of Automata*. Pergamon Press Reprint, 1969.

[7] J. Ashley and L.E.Holloway. Diagnosis of condition systems using causal structure. In *American Control Conference*, volume 1, pages 716–721, 2002.

[8] P. Baroni, G. Lamperti, P. Pogliano, and M. Zanella. Diagnosis of large active systems. *Artif. Intell.*, 110(1):135–183, 1999.

[9] Charlton Barreto, Vaughn Bullard, Thomas Erl, John Evdemon, Diane Jordan, Khanderao Kand, Dieter König, Simon Moser, Ralph Stout, Ron Ten-Hove, Ivana Trickovic, Danny van der Rijn, and Alex Yiu. Web services business process execution

language version 2.0 primer. Technical report, OASIS, May 2007. http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm.

[10] Maurice Beek, Antonio Bucchiarone, and Stefania Gnesi. A survey on service composition approaches: From industrial standards to formal methods. Technical Report 2006TR-15,, Istituto di Scienza e Tecnologie dell'Informazione/IMT Graduate School, Area della Ricerca CNR di Pisa, Via G. Moruzzi 1, 56124 Pisa, Italy, 2006.

[11] Emmanuel Benazera and Louise Travé-Massuyès. Set-theoretic estimation of hybrid system configurations. *Trans. Sys. Man Cyber. Part B*, 39(5):1277–1291, 2009.

[12] Albert Benveniste, Eric Fabre, Claude Jard, and Stefan Haar. Diagnosis of asynchronous discrete event systems, a net unfolding approach. *IEEE Trans. on Automatic Control*, 48:714–727, 2003.

[13] Rene K. Boel and Jan H. van Schuppen. Decentralized failure diagnosis for discrete-event systems with costly communication between diagnosers. In *International Workshop on Discrete Event Systems*, pages 175–181, 2002.

[14] A. Boufaied, A. Subias, and M. Combacau. Chronicle modeling by petri nets for distributed detection of process failures. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 4, Oct. 2002.

[15] Khouloud Boukadi, Chirine Ghedira, Zakaria Maamar, and Djamal Benslimane. Specification and verification of views over composite web services using high level petri-nets. In Jorge Cardoso, José Cordeiro, and Joaquim Filipe, editors, *ICEIS (4)*, pages 107–112, 2007.

[16] Khouloud Boukadi, Chirine Ghedira, Zakaria Maamar, and Hanifa Boucheneb. Specification and verification of views over composite web services using high level petri-nets. Technical Report RR-LIRIS-2006-014, LIRIS UMR 5205 CNRS/INSA de Lyon/Universit¨ Claude Bernard Lyon 1/Université Lumiere Lyon 2/Ecole Centrale de Lyon, 2006.

[17] Maria Paola Cabasino. *Diagnosis and Identification of Discrete Event Systems using Petri Nets*. PhD thesis, University of Cagliari, Italy, march 2009.

[18] J. Cardoso, L.A. Kunzle, and R. Valette. Petri net based reasoning for the diagnosis of dynamic discrete event systems. In *In 6th International Fuzzy Systemes Association World Congress*, page 333¨C336, July 1995.

[19] Thomas Chatain and Claude Jard. Symbolic diagnosis of partially observable concurrent systems. In David de Frutos-Escrig and Manuel Nú nez, editors, *FORTE*, volume 3235 of *Lecture Notes in Computer Science*, pages 326–342, Madrid Spain, September 2004. Springer.

[20] Thomas Chatain and Claude Jard. Models for the supervision of web services orchestration with dynamic changes. In *Advanced Industrial Conference on Telecommunications / Service Assurance with Partial and Intermittent Resources Conference / E-Learning on Telecommunications Workshop*, pages 446–451, Lisbon, Portugal, 2005. IEEE CS Press.

[21] Thomas Chatain and Claude Jard. Models for the supervision of web services orchestration with dynamic changes. In *AICT/SAPIR/ELETE*, pages 446–451. IEEE, IEEE Computer Society, 2005.

[22] Ludmila Cherkasova, Al Davis, Vadim E. Kotov, and Tomas Rokicki. Colored petri net methods for performance analysis of scalable high-speed interconnects. In *MASCOTS '94: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, pages 401–402, Washington, DC, USA, 1994. IEEE Computer Society.

[23] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. Uddi spec technical committee draft. Technical report, OASIS, October 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm.

[24] Luca Console, Claudia Picardi, and Marina Ribaudo. Process algebras for systems diagnosis. *Artificial Intelligence*, 142(1):19–51, November 2002.

[25] M.-O. Cordier, P. Dague, F. Levy, J. Montmain, M. Staroswiecki, and L. Trave-Massuyes. Conflicts versus analytical redundancy relations: a comparative analysis of the model based diagnosis approach from the artificial intelligence and automatic control perspectives. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions*, 34(5):2163–2177, Oct. 2004.

[26] Marie-Odile Cordier and Alban Grastien. Exploiting independence in a decentralised and incremental approach of diagnosis. In Manuela M. Veloso, editor, *IJCAI*, pages 292–297, Hyderabad, India, 2007. Morgan Kaufmann.

[27] R. Scott Cost, Ye Chen, Tim Finin, Yannis Labrou, and Yun Peng. Modeling agent conversations with colored petri nets, 1999.

[28] R. Scott Cost, Ye Chen, Tim Finin, Yannis K Labrou, and Yun Peng. *Using Colored Petri Nets for Conversation Modeling*, volume 1916 of *Lecture Notes in AI*, pages 178–192. Springer-Verlag, September 2000.

[29] Matthew Daigle, Xenofon Koutsoukos, and Gautam Biswas. A discrete event approach to diagnosis of continuous systems. In *Computer Science, Vanderbilt University*, pages 259–266, 2007.

[30] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artif. Intell.*, 24(1-3):347–410, 1984.

[31] de Kleer and J. Kurien. Fundamentals of model-based diagnosis. In *IFAC-SafeProcess*, Washington (USA), 2003. Hindawi Publishing Corp.

[32] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses and systems. *Artificial Intelligence*, 56(2-3):197 – 222, 1992.

[33] Rami Debouk, Stéphane Lafortune, and Demosthenis Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete event systems. *Journal of Discrete Event Dynamical Systems: Theory and Application*, 10:33–86, 2000.

[34] Rami Debouk, Stéphane Lafortune, and Demosthenis Teneketzis. On the effect of communication delays in failure diagnosis of decentralized discrete event systems. *Discrete Event Dynamic Systems*, 13(3):263–289, 2003.

[35] E. Fabre, A. Benveniste, and C. Jard. Distributed diagnosis for large discrete event dynamic systems. In *15th IFAC World Congress, Barcelona*, July 2002.

[36] Eric Fabre and Albert Benveniste. Partial order techniques for distributed discrete event systems: Why you cannot avoid using them. *Discrete Event Dynamic Systems*, 17(3):355–403, 2007.

[37] Eric Fabre, Albert Benveniste, Stefan Haar, and Claude Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15(1):33–84, 2005.

[38] A. Ferrara. Web services: a process algebra approach. In *International Conference of Service Oriented Computing*, pages 242–251, NY, USA, 2004. ACM.

[39] Emilio García, Antonio Correcher Salvador, Francisco Morant, Eduardo Quiles Cucarella, and Ramón Blasco Giménez. Modular fault diagnosis based on discrete event systems. *Discrete Event Dynamic Systems*, 15(3):237–256, 2005.

[40] Michael R. Genesereth. The use of design descriptions in automated diagnosis. *Artif. Intell.*, 24(1-3):411–436, 1984.

[41] A. Giua and C. Seatzu. Fault detection for discrete event systems using petri nets with unobservable transitions. In *44th Int. Conf. on Decision and Control and European Control Conference*, pages 6323–6328, Seville, Spain, December 2005.

[42] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, and H. Nielsen. Simple object access protocol (soap) 1.1. Technical report, World Wide Web Consortium, may 2000. http://www.w3.org/TR/SOAP/.

[43] Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. Soap version 1.2. Technical report, W3C, July 2001. http://www.w3.org/TR/2001/WD-soap12-20010709/.

[44] Xavier Le Guillou, Marie-Odile Cordier, Sophie Robin, and Laurence Rozé. Chronicles for on-line diagnosis of distributed systems. Research report, INARIA, 2008.

[45] Serge Haddad, Tarek Melliti, Patrice Moreaux, and Sylvain Rampacek. Modelling web services interoperability. In *International Conference on Information Systems*, pages 287–295, Virginia, USA, 2004. INSTICC.

[46] Christoforos N. Hadjicostis and George C. Verghese. Monitoring discrete event systems using petri net embeddings. In *Proceedings of the 20th International Conference on Application and Theory of Petri Nets*, pages 188–207, London, UK, 1999. Springer-Verlag.

[47] C.N. Hadjicostis and G.C. Verghese. Power system monitoring based on relay and circuit breaker information. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 3, pages 197–200 vol. 2, May 2001.

[48] Rachid Hamadi and Boualem Benatallah. A petri net-based model for web service composition. In *Australasian Database Conference*, pages 191–200, Adelaide, Australia, 2003. ACM.

[49] Walter Hamscher, Luca Console, and Johan de Kleer. *Readings in model-based diagnosis*. Morgan Kaufmann, San Francisco, USA, 1992.

[50] Xu Han, Zhongzhi Shi, Wenjia Niu, Fen Lin, and Donglei Zhang. An approach for diagnosing unexpected faults in web service flows. *Grid and Cooperative Computing, International Conference on*, 0:61–66, 2009.

[51] Guy Helmer, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, Yanxin Wang, Xia Wang, and Natalia Stakhanova. Software fault tree and coloured petri net based specification, design and implementation of agent-based intrusion detection systems. *Int. J. Inf. Comput. Secur.*, 1(1/2):109–142, 2007.

[52] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *International Conference on Business Process Management*, pages 220–235, Nancy, France, 2005. Springer-Verlag.

[53] Michael W. Hofbaur and Brian C. Williams. Mode estimation of probabilistic hybrid systems. In *HSCC '02: Proceedings of the 5th International Workshop on Hybrid Systems: Computation and Control*, pages 253–266, London, UK, 2002. Springer-Verlag.

[54] Kurt Jensen, Lars Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3):213–254, June 2007.

[55] Shengbing Jiang and Ratnesh Kumar. Failure diagnosis of discrete event systems with linear-time temporal logic fault specifications. In *IEEE Transactions on Automatic Control*, pages 128–133, 2001.

[56] G. Jiroveanu and R. K. Boel. A distributed approach for fault detection and diagnosis based on time petri nets. *Math. Comput. Simul.*, 70(5):287–313, 2006.

[57] G. Jiroveanu and R.K. Boel. Distributed diagnosis of large interacting systems. In *In 16th International Workshop on Principles of Diagnosis (DX¡¯05)*, Monterrey, CA, USA, 2005.

[58] George Jiroveanu. *Fault diagnosis for large Petri nets*. PhD thesis, Belguim, sep 2006.

[59] George Jiroveanu and Rene Boel. Distributed contextual diagnosis for very large systems. In *Proceedings of WODES2004*, pages 1–8, sep 2004. InternalNote: Submitted by: rene.boel@ugent.be.

[60] Panagiotis Katsaros, Vasilis Odontidis, and Maria Gousidou-Koutita. Colored petri net based model checking and failure analysis for e-commerce protocols. In *Dept. of Computer Science, University of Aarhus*, pages 267–283, 2005.

[61] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. Web services choreography description language version 1.0. Technical report, W3C, November 2005. http://www.w3.org/TR/ws-cdl-10/.

[62] Samir Malpathak Kazuhiro Saitou and Helge Qvam. Robust design of flexible manufacturing systems using, colored petri net and genetic algorithm. *Journal of Intelligent Manufacturing*, 13(5):149 – 176, 2002.

[63] Robert M. Keller. Formal verification of parallel programs. *Commun. ACM*, 19(7):371–384, 1976.

[64] K.Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use*. Springer, USA, 1997.

[65] Johan. De Kleer. Local methods for localizing faults in electronical circuits. *MIT-AI Memo*, page 394, 1976.

[66] Xenofon Koutsoukos, Feng Zhao, Horst Haussecker, Jim Reich, and Patrick Cheung. Fault modeling for monitoring and diagnosis of sensor-rich hybrid systems. In *In Proc. of the 40th IEEE Conference on Decision and Control*, pages 793–801, 2001.

[67] Lars Michael Kristensen, Jens B03k J03rgensen, and Kurt Jensen. Application of coloured petri nets in system development. In *In Lecture on Concurrency and Petri Nets, Jorg Desel, Wolfgang Reisig and Grezegorz Rozenberg (Eds.), Springer, LNCS 3089*, pages 626–685. Springer-Verlag, 2004.

[68] R. Kumar and S. Takai. Inference-based ambiguity management in decentralized decision-making: Decentralized control of discrete event systems. *Automatic Control, IEEE Transactions on*, 52(10):1783 –1794, oct. 2007.

[69] R. Kumar and S. Takai. Inference-based ambiguity management in decentralized decision-making: Decentralized diagnosis of discrete-event systems. *IEEE T. Automation Science and Engineering*, 6(3):479–491, 2009.

[70] Jensen Kurt and Rozenberg Grzegorz, editors. *High-level Petri nets: theory and application*. Springer-Verlag, London, UK, 1991.

[71] G. Lamperti and M. Zanella. Diagnosis of active systems – principles and techniques. 741, 2003.

[72] D. Lefebvre. Firing sequences estimation in vector space over z_3 for ordinary petri nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 38(6):1–1336, 2008.

[73] D. Lefebvre and C. Delherm. Diagnosis with causality relationships and directed paths in petri net models. In *IFAC World Congress*, Prague, Czech Republic, 2005.

[74] D. Lefebvre and Abdellah El Moudni. Firing and enabling sequences estimation for timed petri nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 31(3):153–162, 2001.

[75] Dimitri Lefebvre. Sensoring and diagnosis of des with petri net models. *Fault Detection, Supervision and Safety of Technical Processes*, 6(1):1213 – 1218, 2007.

[76] Dimitri Lefebvre. *Diagnosis of Discrete Event Systems with Petri Nets*. I-Tech Education and Publishing, Feb 2008.

[77] Xitong Li, Yushun Fan, Stuart Madnick, and Quan Z. Sheng. A pattern-based approach to protocol mediation for web services composition. *Information and Software Technology*, 52(3):304 – 323, 2010.

[78] Yingmin Li, Tarek Melliti, and Philippe Dague. Modeling bpel ws for diagnosis: towards self-healing ws. In *WEBIST*, pages 795–803, Bacelone, Spein, 2007. IEEE C.S.

[79] N. Lohmann. *A feature-complete petri net semantics for ws-bpel 2.0 and its compiler bpel2owfn*, volume 4937 of *Lecture Notes in Computer Science*, pages 77–91. Springer, Berlin/Heidelberg, frist edition, April 2008. This is a full INBOOK entry.

[80] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting ws-bpel processes using exible model generation. *Data and Knowledge Engineering*, 64(1):38–54, January 2008.

[81] Irina A. Lomazova. On proving large distributed systems: Petri net modules verification. In *PaCT '97: Proceedings of the 4th International Conference on Parallel Computing Technologies*, pages 70–75, London, UK, 1997. Springer-Verlag.

[82] J. Lunze and P. Supavatanakul. Diagnosis of timed automata with an application to industrial actuators. *Integrated Computer Aided Engineering*, 11(1):25–36, 2004.

[83] Jan Lunze. Fault diagnosis of discretely controlled continuous systems by means of discrete-event models. *Discrete Event Dynamic Systems*, 18(2):181–210, 2008.

[84] Marko Mäkelä. Maria: Modular reachability analyzer for high-level Petri nets. In *The 5th Workshop on Discrete Event Systems (WODES 2000)*, pages 477–478, Ghent, Belgium, August 2000. Kluwer Academic Publishers, Boston, MA, USA.

[85] Massimo Mecella, Francesco Parisi Presicce, and Barbara Pernici. Modeling e-service orchestration through petri nets. In *VLDB Workshop on Technologies for E-Services*, pages 38–47, Hong Kong, China, 2002. Springer-Verlag.

[86] M. Merz, D. Moldt, K. Müller, and W. Lamersdorf. Workflow modeling and execution with coloured petri nets in cosm. 1994.

[87] Nielsen Mogens, Plotkin Gordon D., and Winskel Glynn. Petri nets, event structures and domains. In *Proceedings of the International Sympoisum on Semantics of Concurrent Computation*, pages 266–284, London, UK, 1979. Springer-Verlag.

[88] S. Narasimhan and G. Biswas. Model-based diagnosis of hybrid systems. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 37(3):348–361, May 2007.

[89] E. Néth, I.T. Cameron, and K.M. Hangos. Diagnostic goal driven modelling and simulation of multiscale process systems. *Computers Chemical Engineering*, 29(4):783 – 796, 2005. Control of Multiscale and Distributed Process Systems.

[90] OASIS. Bpel 2.0 specification. `http://docs.oasis-open.org/wsbpel/2.0/`.

[91] Marie odile Cordier, Christine Largouët, and Christine Largou. Using model-checking techniques for diagnosing discrete-event systems. In *Workshop on Principles of Diagnosis*, pages 39–46, 2001.

[92] Chun Ouyang, Eric Verbeek, Stephen Breutel, Marlon Dumas, H. M. Arthur, and Ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2-3):162–198, July 2007.

[93] C.M. Özveren and A.S. Willsky. Observability of discrete event dynamic systems. *IEEE Trans. on Aut. Control*, 35(7):797–806, July 1990.

[94] Charles S. Peirce and Eisele Carolyn. *Historical perspectives on Peirce's logic of science : a history of science / edited by Carolyn Eisele*. Mouton Publishers, Berlin ; New York :, 1985.

[95] Chris Peltz. Web services orchestration and choreography. *Computer*, 36:46–52, 2003.

[96] Y. Pencolé and M.-O. Cordier. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence*, 164(1-2):121–170, May 2005.

[97] Yannick Pencolé. Decentralized diagnoser approach: application to telecommunication networks. In *International Workshop on Principles of Diagnosis*, Michoacen, Mexico, 2000. Non.

[98] Yannick Pencole, Marie-Odile Cordier, and Laurence Roze. Incremental decentralized diagnosis approach for the supervision of a telecommunication network. In *IEEE Conference on Decision and Control*, Las Vegas, Nevada, USA, 2002. IEEE.

[99] C. A. Petri. Concepts of net theory. In *MFCS*, pages 137–146, 1973.

[100] Luigi Portinale. Petri net reachability analysis meets model-based diagnostic problem solving. In *In Proceedings IEEE International Conference on Systems, Man and Cybernetics*, pages 2712–2717, 1995.

[101] Kan-John Priscilla and Grastien Alban. Local consistency and junction tree for diagnosis of discrete-event systems. In *Proceeding of the 2008 conference on ECAI 2008*, pages 209–213, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.

[102] Gregory Provan. A model-based diagnosis framework for distributed systems, May 2002.

[103] W. Qiu and R. Kumar. Decentralized failure diagnosis of discrete event systems. *IEEE Transactions on Systems, Man Cybernetics Part A*, 36(2):384–395, 2005.

[104] W. Qiu and R. Kumar. Distributed failure diagnosis under bounded delay using immediate observation passing protocol. In *American Control Conference*, Poland, 2005.

[105] Wenbin Qiu and R. Kumar. A new protocol for distributed diagnosis. In *American Control Conference*, June 2006.

[106] P. Ramadge and W. Wonham. *Discrete event systems: concepts and basic results*. Mouton Publishers, Berlin ; New York :, 1985.

[107] Peter J. Ramadge. Observability of discrete event systems. In *Decision and Control, 1986 25th IEEE Conference on*, volume 25, pages 1108–1112, Dec. 1986.

[108] A. Ramirez-Trevino, E. Ruiz-Beltran, I. Rivera-Rangel, and E. Lopez-Mellado. Online fault diagnosis of discrete event systems. a petri net-based approach. *IEEE TRANSACTIONS ON AUTOMATION SCIENCE AND ENGINEERING*, 4(1):31–39, 2007.

[109] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, April 1987.

[110] Armen Aghasaryan Ren, Eric Fabre, Claude Jard, and Albert Benveniste. A petri net approach to fault detection and diagnosis in distributed systems. In *IEEE Conference on Decision and Control*, pages 702–731, San Diego, CA, 1997. IEEE CS Press.

[111] S. L. Ricker and K. Rudie. Decentralized failure diagnosis with asynchronous communication between supervisors. Technical report, In Proc. of the IEEE Conference on Decision and Control (CDC, 2001.

[112] L. Rodriguez, E. Garcia, F. Morant, A. Correcher, and E. Quiles. *Fault diagnosis for complex systems using Coloured Petri Nets, Petri Nets Applications*. Pawel Pawlewski, first edition, 2010.

[113] Yu Ru and Christoforos N. Hadjicostis. Fault diagnosis in discrete event systems modeled by partially observed petri nets. *Discrete Event Dynamic Systems*, 19(4):551–575, 2009.

[114] G. Salaun, L. Bordeaux, and M. Schaerf. Describing and reasoning on web services using process algebra. In *International Conference on Web Services*, page 43, California, USA, 2004. IEEE.

[115] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, sep 1995.

[116] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D.C. Teneketzis. Failure diagnosis using discrete-event models. *Control Systems Technology, IEEE Transactions on*, 4(2):105–124, Mar 1996.

[117] Karsten Schmidt and Christian Stahl. A petri net semantic for bpel4ws - validation and application. In *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets*, pages 1–6, University of Paderborn, Germany, 2004. University of Paderborn.

[118] S.Genc and S.Lafortune. Distributed diagnosis of place-bordered petri nets. *Automation Science and Engineering, IEEE Transactions*, 4(2):206–219, April 2005.

[119] Robert M. Shapiro. Validation of a vlsi chip using hierarchical colored petri nets. *Microelectronics Reliability*, 31(4):607 – 625, 1991.

[120] S.Lafortune, D. Teneketzis, M. Sampath, R. Sengupta, and K. Sinnamohideen. Failure diagnosis of dynamic systems: an approach based on discrete event systems. In *American Control Conference, 2001. Proceedings of the 2001*, volume 3, pages 2058–2071 vol.3, 2001.

[121] V.S. Srinivasan and M.A. Jafari. Fault detection/monitoring using timed petri nets. *IEEE Transactions On Systems Managment and Cybernetics*, 23(4):1155¨C1162, 1994.

[122] A. Spiteri Staines. A compact colored petri net model for fault diagnosis and recovery in embedded and control systems. *INTERNATIONAL JOURNAL OF COMPUTERS*, 3(2):222–229, 2009.

[123] R. Su and W. M. Wonham. Hierarchical fault diagnosis for discrete-event systems under global consistency. *Discrete Event Dynamic Systems*, 16(1):39–70, 2006.

[124] R. Su, W. M. Wonham, J. Kurien, and X. Koutsoukos. Distributed diagnosis for qualitative systems. In *WODES '02: Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES'02)*, page 169, Washington, DC, USA, 2002. IEEE Computer Society.

[125] R. Su and W.M. Wonham. A model of component consistency in distributed diagnosis. In *Proc.IFAC Workshop on Discrete Event Systems (WODES'04)*, pages 427–432, Reims, France, 2004.

[126] P. Supavatanakul, J. Lunze, V. Puig, and J. Quevedo. Diagnosis of timed automata: Theory and application to the damadics actuator benchmark problem. *Control Engineering Practice*, 14(6):609–619, 2006.

[127] A. Szücs, G. Gerzson, and KM. Hangos. An intelligent diagnostic system based on petri nets. *COMPUTERS CHEMICAL ENGINEERING*, 22(9):1335–1344, 1998.

[128] Louise Travé-Massuyès, Marie Odile Cordier, and Xavier Pucel. Comparing diagnosability in continuous and discrete-event systems. In *Proceedings of the 6th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes (SAFEPROCESS'2006)*, pages 1231–1236, Beijing, P.R. China, August-September 2006. IFAC.

[129] Stavros Tripakis. Fault diagnosis for timed automata. In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT*, volume 2469 of *Lecture Notes in Computer Science*, pages 205–224, 2002.

[130] Edward Tsang. Foundations of constraint satisfaction, 1993.

[131] T. Ushio, L Onishi, and K. Okuda. Fault detection based on petri net models with faulty behaviors. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics*, pages 113–118, San Diego, USA, October 1998.

[132] F. van Breugel and K. Mariya. Dead-path-elimination in bpel4ws. In *International Conference on Application of Concurrency to System Design*, pages 192–201, Turku, Finland, 2005. IEEE CS Press.

[133] M. Viroli. Analyzing interacting ws-bpel processes using exible model generation. *Electronic notes in theoretical computer science*, 105:51–71, December 2004.

[134] Y. Wang, T.S. Yoo, and S. Lafortune. New results on decentralized diagnosis of discrete-event systems. In *Annual Allerton Conference*, 2004.

[135] Yin Wang, Tae-Sic Yoo, and Stéphane Lafortune. Diagnosis of discrete event systems using decentralized architectures. *Discrete Event Dynamic Systems*, 17(2):233–263, 2007.

[136] Brian C. Williams, P. Pandurang Nayak, and Urang Nayak. A model-based approach to reactive self-configuring systems. In *In Proceedings of AAAI-96*, pages 971–978, 1996.

[137] Petri Nets World. Pn tool database. `http://www.informatik.uni-hamburg.de`.

[138] WSDIAMOND. Wsdiamond project. `http://wsdiamond.di.unito.it`.

[139] He xuan Hu, Anne lise Gehin, and Mireille Bayart. A merge method for decentralized discrete-event fault diagnosis. *Internet Monitoring and Protection, International Conference on*, 0:119–124, 2008.

[140] Yuhong Yan. *Description Language and Formal Methods for Web Service Process Modeling*. M.E Sharpe Inc., Armonk USA, 2008.

[141] Yuhong Yan and Philippe Dague. Monitoring and diagnosing orchestrated web service process web services. In *International Conference on Web Services*, pages 9–13, Utah, USA, 2007. IEEE C.S.

[142] Yuhong Yan, Philippe Dague, Yannick Pencolé, and Marie-Odile Cordier. A model-based approach for diagnosing fault in web service processes. *Int. J. Web Service Res.*, 6(1):87–110, 2009.

[143] YanPing Yang, QingPing Tan, and Yong Xiao. Verifying web services composition based on hierarchical colored petri nets. In *workshop on Interoperability of Heterogeneous Information Systems*, pages 47–54, Bremen, Germany, 2005. ACM.

[144] Lina Ye and Philippe Dague. Decentralized diagnosis for bpel web services. In *WEBIST*, pages 283–287, Portugal, 2008. INSTICC.

[145] Xiaochuan Yi and Krys J. Kochut. A cp-nets-based design and verification framework for web services composition. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 756, Washington, DC, USA, 2004. IEEE Computer Society.

[146] S. Hashtrudi Zad, R.H. Kwong, and W.M. Wonham. Fault diagnosis in discrete-event systems: Framework and model reduction. In *in Proc. 1998 IEEE Conference on Decision and Control (CDC¡¯98*, pages 3769–3774, 1998.

[147] S. Hashtrudi Zad, R.H. Kwong, and W.M. Wonham. Fault diagnosis in finite-state automata and timed discrete-event systems. In *In 38th IEEE Conference on Decision and Control*, 1999.

[148] D. A. Zaitsev. Functional petri nets. *Universite Paris-Dauphine, Cahier du Lamsade 224*, pages 62–pp, April 2005. http://www.lamsade.dauphine.fr/cahiers.html.

[149] D. A. Zaitsev. Solving the fundamental equation of petri net in the process of composition of functional subnet. *Artificial Intelligence, no. 1, 2005*, pages 59–68, 2005. In Russian.

[150] Zhao-Li Zhang, Fan Hong, and Hai-Jun Xiao. A colored petri net-based model for web service composition. *Journal of Shanghai University (English Edition)*, 105(4):323–329, 2008.

Appendix A

The foodshop file list

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
BPEL Process Definition
Edited using ActiveBPEL(tm) Designer Version 2.1.0 (http://www.active-endpoints.com)
-->
<process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/" xmlns:abx="http://www.activebpel.org/bpel/extension" xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/" xmlns:ext="http://www.activebpel.org/2.0/bpel/extension" xmlns:ns1="urn:ShopWS/wsdl" xmlns:ns2="urn:/WarehouseServer/" xmlns:ns3="urn:/RealSupplierServer/" xmlns:ns4="urn:SupplierWS/types" xmlns:ns5="urn:ShopWS/types" xmlns:ns6="urn:WarehouseWS/types" xmlns:ns7="urn:LocalShopWS1/wsdl" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="Shop" suppressJoinFailure="yes" targetNamespace="http://Shop">
  <partnerLinks>
    <partnerLink myRole="ShopProvider" name="Shop-Shop2HumanClient" partnerLinkType="ns1:Shop-Shop2HumanClient"/>
    <partnerLink myRole="WHCallBack" name="Shop-Shop2WH" partnerLinkType="ns1:Shop-Shop2WH" partnerRole="WarehouseProvider"/>
    <partnerLink myRole="RealSupplierCallBack" name="Shop-Shop2RealSupplier" partnerLinkType="ns1:Shop-Shop2RealSupplier" partnerRole="RealSupplierProvider"/>
    <partnerLink name="Shop-Shop2LocalShopService" partnerLinkType="ns1:Shop-Shop2LocalShopService" partnerRole="LocalShopServiceProvider"/>
  </partnerLinks>
  <variables>
```

```
      <variable messageType=”ns1:WHAnswerMSG” name=”WHAnswerMSG”/>
15    <variable messageType=”ns1:RealSupplierAnswerMSG” name=”
          RealSupplierAnswerMSG”/>
      <variable messageType=”ns1:ShopWSSEI_receiveOrder” name=”
          ShopWSSEI_receiveOrder”/>
17    <variable messageType=”ns1:ShopWSSEI_reply2Client” name=”
          ShopWSSEI_reply2Client”/>
      <variable messageType=”ns3:SupplierWSSEI_checkAvailReserve” name=”
          SupplierWSSEI_checkAvailReserve”/>
19    <variable messageType=”ns2:WarehouseWSSEI_checkAvailable” name=”
          WarehouseWSSEI_checkAvailable”/>
      <variable messageType=”ns3:SupplierWSSEI_unReserve” name=”
          SupplierWSSEI_unReserve”/>
21    <variable messageType=”ns3:SupplierWSSEI_requestSupply” name=”
          SupplierWSSEI_requestSupply”/>
      <variable messageType=”ns2:WarehouseWSSEI_confirmOrder” name=”
          WarehouseWSSEI_confirmOrder”/>
23    <variable messageType=”ns2:WarehouseWSSEI_unReserve” name=”
          WarehouseWSSEI_unReserve”/>
      <variable messageType=”ns7:receiveOrder1Request” name=”
          receiveOrder1Request”/>
25    <variable messageType=”ns7:receiveOrder1Response” name=”
          receiveOrder1Response”/>
      <variable messageType=”ns7:CalculateTotalPriceRequest” name=”
          CalculateTotalPriceRequest”/>
27    <variable messageType=”ns7:CalculateTotalPriceResponse” name=”
          CalculateTotalPriceResponse”/>
      <variable name=”TempV2” type=”xsd:boolean”/>
29    <variable name=”TempV3” type=”xsd:boolean”/>
      <variable messageType=”ns3:ConfirmMSG” name=”ConfirmMSG”/>
31    <variable messageType=”ns1:ExternalProblemMSG” name=”ExternalProblemMSG”/>
      <variable messageType=”ns1:ExternalProblemsMSG-RealSupplier” name=”
          ExternalProblemsMSG-RealSupplier”/>
33    <variable messageType=”ns7:GetItemNameListRequest” name=”
          GetItemNameListRequest”/>
      <variable messageType=”ns7:GetItemNameListResponse” name=”
          GetItemNameListResponse”/>
35  </variables>
    <correlationSets>
37    <correlationSet name=”CS1” properties=”ns1:StCorrelation”/>
    </correlationSets>
```

```
39  <sequence>
      <receive createInstance="yes" name="Rec_From_User" operation="receiveOrder
          " partnerLink="Shop-Shop2HumanClient" portType="ns1:ShopPT" variable="
          ShopWSSEI_receiveOrder">
41      <correlations>
          <correlation initiate="yes" set="CS1"/>
43      </correlations>
      </receive>
45    <assign>
        <copy>
47        <from part="parameters" query="/ns5:receiveOrder/PID" variable="
              ShopWSSEI_receiveOrder"/>
          <to part="PID" variable="receiveOrder1Request"/>
49      </copy>
        <copy>
51        <from part="parameters" query="/ns5:receiveOrder/String_2" variable=
              "ShopWSSEI_receiveOrder"/>
          <to part="ItemList" variable="receiveOrder1Request"/>
53      </copy>
      </assign>
55    <invoke inputVariable="receiveOrder1Request" name="
          Inv_SplitOrder_on_ShopLocalService" operation="receiveOrder1"
          outputVariable="receiveOrder1Response" partnerLink="Shop-
          Shop2LocalShopService" portType="ns7:LocalShopPT"/>
      <assign name="Assign-prepareMSGforSupplier">
57      <copy>
          <from part="PID" variable="receiveOrder1Response"/>
59        <to part="PID" variable="SupplierWSSEI_checkAvailReserve"/>
        </copy>
61      <copy>
          <from part="PerishableList" variable="receiveOrder1Response"/>
63        <to part="ItemList" variable="SupplierWSSEI_checkAvailReserve"/>
        </copy>
65      <copy>
          <from part="parameters" query="/ns5:receiveOrder/CustInfo_1"
              variable="ShopWSSEI_receiveOrder"/>
67        <to part="CustInfo" variable="SupplierWSSEI_checkAvailReserve"/>
        </copy>
69    </assign>
      <assign name="Assign-prepareMSGforWarehouse">
71      <copy>
```

```
                <from part="PID" variable="receiveOrder1Response"/>
73              <to part="PID" variable="WarehouseWSSEI_checkAvailable"/>
            </copy>
75          <copy>
                <from part="UnPerishableList" variable="receiveOrder1Response"/>
77              <to part="parameters" query="/ns6:checkAvailReserve/String_1"
                    variable="WarehouseWSSEI_checkAvailable"/>
            </copy>
79          <copy>
                <from part="parameters" query="/ns5:receiveOrder/CustInfo_1"
                    variable="ShopWSSEI_receiveOrder"/>
81              <to part="parameters" query="/ns6:checkAvailReserve/CustInfo"
                    variable="WarehouseWSSEI_checkAvailable"/>
            </copy>
83      </assign>
        <scope variableAccessSerializable="no">
85          <eventHandlers>
                <onMessage operation="ExternalProblemsManagement" partnerLink="Shop-
                    Shop2WH" portType="ns1:WHcallbackPT" variable="
                    ExternalProblemMSG">
87                  <correlations>
                        <correlation set="CS1"/>
89                  </correlations>
                    <ext:suspend name="Suspend_on_WH_Request"/>
91              </onMessage>
                <!-- MSG from Supplier -->
93              <onMessage operation="ExternalProblemsManagement-RealSupplier"
                    partnerLink="Shop-Shop2RealSupplier" portType="
                    ns1:RealSuppliercallbackPT" variable="ExternalProblemsMSG-
                    RealSupplier">
                    <correlations>
95                      <correlation set="CS1"/>
                    </correlations>
97                  <ext:suspend name="Suspend_on_RealSupplierRequest"/>
                </onMessage>
99          </eventHandlers>
            <sequence>
101             <flow>
                    <sequence>
```

```
103           <invoke inputVariable="SupplierWSSEI_checkAvailReserve" name="
                  Inv_CheckAvailaReserve_on_Supplier" operation="
                  checkAvailReserve" partnerLink="Shop-Shop2RealSupplier"
                  portType="ns3:RealSupplierServer">
                <correlations>
105               <correlation pattern="out" set="CS1"/>
                </correlations>
107           </invoke>
              <receive name="Rec_SupplierAnswer" operation="
                  receiveRealSupplierAnswer" partnerLink="Shop-
                  Shop2RealSupplier" portType="ns1:RealSuppliercallbackPT"
                  variable="RealSupplierAnswerMSG">
109             <correlations>
                  <correlation set="CS1"/>
111             </correlations>
              </receive>
113         </sequence>
            <sequence>
115           <invoke inputVariable="WarehouseWSSEI_checkAvailable" name="
                  Inv_CheckAvailable_on_WH" operation="checkAvailable"
                  partnerLink="Shop-Shop2WH" portType="ns2:WarehouseServer">
                <correlations>
117               <correlation pattern="out" set="CS1"/>
                </correlations>
119           </invoke>
              <receive name="Rec_WHAnswerMSG" operation="receiveAnswer"
                  partnerLink="Shop-Shop2WH" portType="ns1:WHcallbackPT"
                  variable="WHAnswerMSG">
121             <correlations>
                  <correlation set="CS1"/>
123             </correlations>
              </receive>
125         </sequence>
          </flow>
127       <assign>
            <copy>
129           <from part="Availability" variable="RealSupplierAnswerMSG"/>
              <to variable="TempV2"/>
131         </copy>
            <copy>
133           <from part="Availability" variable="WHAnswerMSG"/>
```

```
                <to variable="TempV3"/>
135           </copy>
            </assign>
137         <switch>
              <case condition="(bpws:getVariableData('TempV2') and
                  bpws:getVariableData('TempV3'))">
139             <sequence>
                  <assign>
141                 <copy>
                      <from part="parameters" query="/ns5:receiveOrder/PID"
                            variable="ShopWSSEI_receiveOrder"/>
143                   <to part="PID" variable="WarehouseWSSEI_confirmOrder"
                          />
                    </copy>
145                 <copy>
                      <from part="parameters" query="/ns5:receiveOrder/
                          CustInfo_1" variable="ShopWSSEI_receiveOrder"/>
147                   <to part="CustInfo_2" variable="
                          WarehouseWSSEI_confirmOrder"/>
                    </copy>
149                 <copy>
                      <from part="UnPerishableList" variable="
                          receiveOrder1Response"/>
151                   <to part="String_1" variable="
                          WarehouseWSSEI_confirmOrder"/>
                    </copy>
153               </assign>
                  <invoke inputVariable="WarehouseWSSEI_confirmOrder" name="
                      Inv_ConfirmOrder_on_WH" operation="confirmOrder"
                      partnerLink="Shop-Shop2WH" portType="
                      ns2:WarehouseServer">
155                 <correlations>
                      <correlation pattern="out" set="CS1"/>
157                 </correlations>
                  </invoke>
159               <assign>
                    <copy>
161                   <from part="parameters" query="/ns5:receiveOrder/
                          CustInfo_1" variable="ShopWSSEI_receiveOrder"/>
                      <to part="CustInfo" variable="ConfirmMSG"/>
163                 </copy>
```

```
                    <copy>
165                   <from part="parameters" query="/ns5:receiveOrder/PID"
                          variable="ShopWSSEI_receiveOrder"/>
                      <to part="PID" variable="ConfirmMSG"/>
167                 </copy>
                  </assign>
169               <invoke inputVariable="ConfirmMSG" name="
                      Inv_Confirm_on_RealSupplier" operation="Confirm"
                      partnerLink="Shop-Shop2RealSupplier" portType="
                      ns3:RealSupplierServer"/>
                  <assign>
171                 <copy>
                      <from expression="concat( bpws:getVariableData('
                          receiveOrder1Response', 'PerishableList') ,&quot
                          ;,&quot; ,bpws:getVariableData('
                          receiveOrder1Response', 'UnPerishableList') )"/>
173                   <to part="ItemList" variable="
                          CalculateTotalPriceRequest"/>
                    </copy>
175                 <copy>
                      <from part="parameters" query="/ns5:receiveOrder/PID"
                          variable="ShopWSSEI_receiveOrder"/>
177                   <to part="PID" variable="CalculateTotalPriceRequest"/
                         >
                    </copy>
179               </assign>
                  <invoke inputVariable="CalculateTotalPriceRequest" name="
                      Inv_CalculateTotalPrice_on_ShopLocalService" operation=
                      "CalculateTotalPrice" outputVariable="
                      CalculateTotalPriceResponse" partnerLink="Shop-
                      Shop2LocalShopService" portType="ns7:LocalShopPT"/>
181               <assign name="Prepare_MSG_for_Get_Item_Name">
                    <copy>
183                   <from expression="concat( bpws:getVariableData('
                          receiveOrder1Response', 'PerishableList') ,&quot
                          ;,&quot;, bpws:getVariableData('
                          receiveOrder1Response', 'UnPerishableList') )"/>
                      <to part="getItemNameListRequest" variable="
                          GetItemNameListRequest"/>
185                 </copy>
                  </assign>
```

```
187                     <invoke inputVariable="GetItemNameListRequest" name="Inv-
                            GetItemNameList" operation="GetItemNameList"
                            outputVariable="GetItemNameListResponse" partnerLink="
                            Shop-Shop2LocalShopService" portType="ns7:LocalShopPT"/
                            >
                        <assign>
189                         <copy>
                                <from part="getItemNameListResponse" variable="
                                    GetItemNameListResponse"/>
191                             <to part="IncomingItems" variable="
                                    ShopWSSEI_reply2Client"/>
                            </copy>
193                         <copy>
                                <from part="parameters" query="/ns5:receiveOrder/PID"
                                     variable="ShopWSSEI_receiveOrder"/>
195                             <to part="PID" variable="ShopWSSEI_reply2Client"/>
                            </copy>
197                         <copy>
                                <from expression="concat(&quot;Thanks for the
                                    shopping, total price is &quot;,
                                    bpws:getVariableData('CalculateTotalPriceResponse
                                    ', 'Price') )"/>
199                             <to part="replyMsg" variable="ShopWSSEI_reply2Client"
                                    />
                            </copy>
201                     </assign>
                    </sequence>
203             </case>
                <otherwise>
205                 <sequence>
                        <assign>
207                         <copy>
                                <from part="parameters" query="/ns5:receiveOrder/PID"
                                     variable="ShopWSSEI_receiveOrder"/>
209                             <to part="PID" variable="SupplierWSSEI_unReserve"/>
                            </copy>
211                         <copy>
                                <from part="parameters" query="/ns5:receiveOrder/PID"
                                     variable="ShopWSSEI_receiveOrder"/>
213                             <to part="PID" variable="WarehouseWSSEI_unReserve"/>
                            </copy>
```

```
215                 <copy>
                        <from part="parameters" query="/ns5:receiveOrder/
                            CustInfo_1" variable="ShopWSSEI_receiveOrder"/>
217                     <to part="parameters" query="/ns4:unReserve/CustInfo"
                             variable="SupplierWSSEI_unReserve"/>
                    </copy>
219                 <copy>
                        <from expression="&quot;prova&quot;"/>
221                     <to part="String_1" variable="
                            WarehouseWSSEI_unReserve"/>
                    </copy>
223                 <copy>
                        <from part="parameters" query="/ns5:receiveOrder/PID"
                             variable="ShopWSSEI_receiveOrder"/>
225                     <to part="parameters" query="/ns4:unReserve/PID"
                            variable="SupplierWSSEI_unReserve"/>
                    </copy>
227             </assign>
                <invoke inputVariable="WarehouseWSSEI_unReserve" name="
                    Inv_Unreserve_on_WH" operation="unReserve" partnerLink=
                    "Shop-Shop2WH" portType="ns2:WarehouseServer">
229                 <correlations>
                        <correlation pattern="out" set="CS1"/>
231                 </correlations>
                </invoke>
233             <invoke inputVariable="SupplierWSSEI_unReserve" name="
                    Inv_Unreserve_on_RealSupplier" operation="unReserve"
                    partnerLink="Shop-Shop2RealSupplier" portType="
                    ns3:RealSupplierServer">
                    <correlations>
235                     <correlation pattern="out" set="CS1"/>
                    </correlations>
237             </invoke>
                <assign>
239                 <copy>
                        <from expression="concat( &quot;Not available items:
                            &quot;,bpws:getVariableData('
                            RealSupplierAnswerMSG', 'unReservedItems') ,&quot
                            ;,&quot;, bpws:getVariableData('WHAnswerMSG', '
                            unReservedItems') )"/>
```

```
241                               <to part="getItemNameListRequest" variable="
                                      GetItemNameListRequest"/>
                              </copy>
243                       </assign>
                          <invoke inputVariable="GetItemNameListRequest" name="Inv-
                              GetItemNameList" operation="GetItemNameList"
                              outputVariable="GetItemNameListResponse" partnerLink="
                              Shop-Shop2LocalShopService" portType="ns7:LocalShopPT"/
                              >
245                       <assign>
                              <copy>
247                               <from expression="&quot;sorry something is not
                                      available&quot;"/>
                                  <to part="replyMsg" variable="ShopWSSEI_reply2Client"
                                      />
249                           </copy>
                              <copy>
251                               <from part="parameters" query="/ns5:receiveOrder/PID"
                                       variable="ShopWSSEI_receiveOrder"/>
                                  <to part="PID" variable="ShopWSSEI_reply2Client"/>
253                           </copy>
                              <copy>
255                               <from part="getItemNameListResponse" variable="
                                      GetItemNameListResponse"/>
                                  <to part="IncomingItems" variable="
                                      ShopWSSEI_reply2Client"/>
257                           </copy>
                          </assign>
259                   </sequence>
                  </otherwise>
261           </switch>
              <wait for="'PT10S'"/>
263       </sequence>
      </scope>
265   <reply name="Reply_to_client" operation="receiveOrder" partnerLink="Shop-
          Shop2HumanClient" portType="ns1:ShopPT" variable="
          ShopWSSEI_reply2Client"/>
  </sequence>
267 </process>
```

Listing A.1: SHOP BPEL service

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <!--
3 BPEL Process Definition
  Edited using ActiveBPEL(tm) Designer Version 2.1.0 (http://www.active-endpoints.
      com)
5 -->
  <process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:ext="http://www.activebpel.org/2.0/bpel/extension" xmlns:ns1="urn:/
      RealSupplierServer/" xmlns:ns2="urn:ShopWS/wsdl" xmlns:ns3="urn:SupplierWS/
      types" xmlns:ns4="urn:SupplierLocWS1/wsdl" xmlns:ns5="urn:/WarehouseServer/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="RealSupplier"
      suppressJoinFailure="yes" targetNamespace="http://RealSupplier">
7   <partnerLinks>
      <partnerLink myRole="RealSupplierProvider" name="RealSupplier-
          RealSupplier2Shop" partnerLinkType="ns1:RealSupplier-RealSupplier2Shop
          " partnerRole="RealSupplierCallBack"/>
9     <partnerLink myRole="RealSupplierProvider" name="RealSupplier-
          RealSupplier2WareHouse" partnerLinkType="ns1:RealSupplier-
          RealSupplier2WareHouse"/>
      <partnerLink name="RealSupplier-RealSupplier2RealSupplierLocalService"
          partnerLinkType="ns1:RealSupplier-
          RealSupplier2RealSupplierLocalService" partnerRole="
          RealSupplierLocalServiceProvider"/>
11  </partnerLinks>
    <variables>
13    <variable messageType="ns2:RealSupplierAnswerMSG" name="
          RealSupplierAnswerMSG"/>
      <variable messageType="ns1:SupplierWSSEI_checkAvailReserve" name="
          SupplierWSSEI_checkAvailReserve"/>
15    <variable messageType="ns1:SupplierWSSEI_requestSupply" name="
          SupplierWSSEI_requestSupply"/>
      <variable messageType="ns1:SupplierWSSEI_unReserve" name="
          SupplierWSSEI_unReserve"/>
17    <variable messageType="ns1:SupplierWSSEI_ShippingRequest" name="
          SupplierWSSEI_ShippingRequest"/>
      <variable messageType="ns4:CheckLocalAvailability1Request" name="
          CheckLocalAvailability1Request"/>
19    <variable messageType="ns4:CheckLocalAvailability1Response" name="
          CheckLocalAvailability1Response"/>
```

```
      <variable messageType="ns4:ArrangeShippingRequest" name="
          ArrangeShippingRequest"/>
21    <variable messageType="ns4:ArrangeShippingResponse" name="
          ArrangeShippingResponse"/>
      <variable messageType="ns1:ConfirmMSG" name="ConfirmMSG"/>
23    <variable messageType="ns4:ReservationRequest" name="ReservationRequest"/>
      <variable messageType="ns4:ReservationResponse" name="ReservationResponse"
          />
25    <variable messageType="ns4:UnReservationRequest" name="
          UnReservationRequest"/>
      <variable messageType="ns4:UnReservationResponse" name="
          UnReservationResponse"/>
27    <variable messageType="ns1:ExternalProblemManagementMSG" name="
          ExternalProblemManagementMSG"/>
      <variable messageType="ns2:ExternalProblemsMSG-RealSupplier" name="
          ExternalProblemsMSG-RealSupplier"/>
29  </variables>
    <correlationSets>
31    <correlationSet name="CS1" properties="ns1:RealSupplierCorrelationSet"/>
    </correlationSets>
33  <sequence>
      <receive createInstance="yes" name="Rec_Request_from_Shop" operation="
          checkAvailReserve" partnerLink="RealSupplier-RealSupplier2Shop"
          portType="ns1:RealSupplierServer" variable="
          SupplierWSSEI_checkAvailReserve">
35      <correlations>
          <correlation initiate="yes" set="CS1"/>
37      </correlations>
      </receive>
39    <assign>
        <copy>
41        <from part="PID" variable="SupplierWSSEI_checkAvailReserve"/>
          <to part="PID" variable="CheckLocalAvailability1Request"/>
43      </copy>
        <copy>
45        <from part="ItemList" variable="SupplierWSSEI_checkAvailReserve"/>
          <to part="parameters" variable="CheckLocalAvailability1Request"/>
47      </copy>
      </assign>
```

```
49      <invoke inputVariable="CheckLocalAvailability1Request" name="
            Inv_CheckAvailability_on_RealSupplierLocalService" operation="
            CheckLocalAvailability1" outputVariable="
            CheckLocalAvailability1Response" partnerLink="RealSupplier-
            RealSupplier2RealSupplierLocalService" portType="ns4:SupplierLocPT"/>
        <assign>
51          <copy>
                <from part="Availability" variable="CheckLocalAvailability1Response"
                    />
53              <to part="Availability" variable="RealSupplierAnswerMSG"/>
            </copy>
55          <copy>
                <from part="UnPerishableList" variable="
                    CheckLocalAvailability1Response"/>
57              <to part="unReservedItems" variable="RealSupplierAnswerMSG"/>
            </copy>
59          <copy>
                <from part="PID" variable="CheckLocalAvailability1Response"/>
61              <to part="PID" variable="RealSupplierAnswerMSG"/>
            </copy>
63      </assign>
        <invoke inputVariable="RealSupplierAnswerMSG" name="
            Inv_SupplierAnswer_on_Shop" operation="receiveRealSupplierAnswer"
            partnerLink="RealSupplier-RealSupplier2Shop" portType="
            ns2:RealSuppliercallbackPT">
65          <correlations>
                <correlation pattern="out" set="CS1"/>
67          </correlations>
        </invoke>
69      <scope variableAccessSerializable="no">
            <faultHandlers>
71              <catchAll>
                    <sequence>
73                      <assign>
                            <copy>
75                              <from part="PID" variable="
                                    SupplierWSSEI_checkAvailReserve"/>
                                <to part="PID" variable="ExternalProblemManagementMSG"/>
77                          </copy>
                            <copy>
79                              <from expression="'General Problem'"/>
```

```
                    <to part="ProblemCode" variable="ExternalProblemsMSG-
                        RealSupplier"/>
81                  </copy>
                </assign>
83              <invoke inputVariable="ExternalProblemsMSG-RealSupplier" name=
                    "Inv-Suspend_on_Shop" operation="
                    ExternalProblemsManagement-RealSupplier" partnerLink="
                    RealSupplier-RealSupplier2Shop" portType="
                    ns2:RealSuppliercallbackPT"/>
                <ext:suspend/>
85          </sequence>
        </catchAll>
87    </faultHandlers>
      <eventHandlers>
89      <onMessage operation="ExternalProblemsManagement" partnerLink="
             RealSupplier-RealSupplier2WareHouse" portType="
             ns1:RealSupplierServer" variable="ExternalProblemManagementMSG">
          <correlations>
91          <correlation set="CS1"/>
          </correlations>
93        <sequence>
            <ext:suspend/>
95        </sequence>
        </onMessage>
97    </eventHandlers>
      <pick>
99      <onMessage operation="unReserve" partnerLink="RealSupplier-
             RealSupplier2Shop" portType="ns1:RealSupplierServer" variable="
             SupplierWSSEI_unReserve">
          <correlations>
101         <correlation set="CS1"/>
          </correlations>
103       <sequence>
            <assign>
105           <copy>
                <from part="PID" variable="
                    SupplierWSSEI_checkAvailReserve"/>
107             <to part="PID" variable="UnReservationRequest"/>
              </copy>
109           <copy>
```

```
                    <from part="CustInfo" variable="
                        SupplierWSSEI_checkAvailReserve"/>
111                 <to part="CustInfo" variable="UnReservationRequest"/>
                  </copy>
113               <copy>
                    <from part="parameters" variable="
                        SupplierWSSEI_unReserve"/>
115                 <to part="parameters" variable="UnReservationRequest"/>
                  </copy>
117             </assign>
                <invoke inputVariable="UnReservationRequest" name="
                    Inv_Unreserve_on_RealSupplier_LocalService" operation="
                    UnReservation" outputVariable="UnReservationResponse"
                    partnerLink="RealSupplier-
                    RealSupplier2RealSupplierLocalService" portType="
                    ns4:SupplierLocPT"/>
119           </sequence>
          </onMessage>
121       <onMessage operation="Confirm" partnerLink="RealSupplier-
              RealSupplier2Shop" portType="ns1:RealSupplierServer" variable="
              ConfirmMSG">
            <correlations>
123           <correlation set="CS1"/>
            </correlations>
125         <sequence>
              <assign>
127             <copy>
                  <from part="parameters" variable="
                      CheckLocalAvailability1Request"/>
129               <to part="parameters" variable="ReservationRequest"/>
                </copy>
131             <copy>
                  <from part="CustInfo" variable="
                      SupplierWSSEI_checkAvailReserve"/>
133               <to part="CustInfo" variable="ReservationRequest"/>
                </copy>
135             <copy>
                  <from part="PID" variable="
                      SupplierWSSEI_checkAvailReserve"/>
137               <to part="PID" variable="ReservationRequest"/>
                </copy>
```

```
139             </assign>
                <invoke inputVariable="ReservationRequest" name="
                    Inv_Reserve_on_REalSupplier_LocalService" operation="
                    Reservation" outputVariable="ReservationResponse"
                    partnerLink="RealSupplier-
                    RealSupplier2RealSupplierLocalService" portType="
                    ns4:SupplierLocPT"/>
141             <receive name="Rec_ShippingReq_from_WH" operation="
                    ArrangeShipping" partnerLink="RealSupplier-
                    RealSupplier2WareHouse" portType="ns1:RealSupplierServer"
                    variable="SupplierWSSEI_ShippingRequest">
                  <correlations>
143                 <correlation set="CS1"/>
                  </correlations>
145             </receive>
                <assign>
147               <copy>
                    <from part="ShippingData" variable="
                        SupplierWSSEI_ShippingRequest"/>
149                 <to part="ItemList" variable="ArrangeShippingRequest"/>
                  </copy>
151               <copy>
                    <from part="PID" variable="
                        SupplierWSSEI_checkAvailReserve"/>
153                 <to part="PID" variable="ArrangeShippingRequest"/>
                  </copy>
155               <copy>
                    <from part="CustInfo" variable="
                        SupplierWSSEI_checkAvailReserve"/>
157                 <to part="CustInfo" variable="ArrangeShippingRequest"/>
                  </copy>
159             </assign>
                <invoke inputVariable="ArrangeShippingRequest" name="
                    Inv_ArrangeShipping_on_RealSupplierLocalService" operation
                    ="ArrangeShipping" outputVariable="ArrangeShippingResponse
                    " partnerLink="RealSupplier-
                    RealSupplier2RealSupplierLocalService" portType="
                    ns4:SupplierLocPT"/>
161           </sequence>
            </onMessage>
163       </pick>
```

```
          </scope>
165     </sequence>
    </process>
```

Listing A.2: SUPPLIER BPEL service

```
  <?xml version="1.0" encoding="UTF-8"?>
2 <!--
  BPEL Process Definition
4 Edited using ActiveBPEL(tm) Designer Version 2.1.0 (http://www.active-endpoints.
      com)
  -->
6 <process xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
      xmlns:ext="http://www.activebpel.org/2.0/bpel/extension" xmlns:ns1="urn:/
      WarehouseServer/" xmlns:ns2="urn:ShopWS/wsdl" xmlns:ns3="urn:/
      LocalSupplierServer/" xmlns:ns4="urn:WarehouseWS/types" xmlns:ns5="urn:/
      RealSupplierServer/" xmlns:ns6="urn:LocalWHWS1/wsdl" xmlns:ns7="
      urn:AeAdminServices" xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="
      Warehouse" suppressJoinFailure="yes" targetNamespace="http://Warehouse">
    <partnerLinks>
8       <partnerLink myRole="LocalSupplierCallBack" name="Warehouse-
            WH2LocalSupplier" partnerLinkType="ns1:Warehouse-WH2LocalSupplier"
            partnerRole="LocalSupplierProvider"/>
        <partnerLink myRole="WarehouseProvider" name="Warehouse-WH2Shop"
            partnerLinkType="ns1:Warehouse-WH2Shop" partnerRole="WHCallback2Shop"/
            >
10      <partnerLink myRole="SupplierCallBack" name="Warehouse2RealSupplier"
            partnerLinkType="ns1:Warehouse" partnerRole="RealSupplierProvider"/>
        <partnerLink name="WH-WH2LocalWHService" partnerLinkType="ns1:WH-
            WH2LocalWHService" partnerRole="LocalWHServiceProvider"/>
12  </partnerLinks>
    <variables>
14      <variable messageType="ns2:WHAnswerMSG" name="WHAnswerMSG"/>
        <variable messageType="ns3:Request" name="Request1"/>
16      <variable messageType="ns1:callbackResponse" name="callbackResponse"/>
        <variable messageType="ns1:WarehouseWSSEI_checkAvailable" name="
            WarehouseWSSEI_checkAvailable"/>
18      <variable name="All_available" type="xsd:boolean"/>
        <variable messageType="ns1:WarehouseWSSEI_unReserve" name="
            WarehouseWSSEI_unReserve"/>
```

```
20        <variable messageType="ns1:WarehouseWSSEI_confirmOrder" name="
              WarehouseWSSEI_confirmOrder"/>
          <variable messageType="ns5:SupplierWSSEI_ShippingRequest" name="
              SupplierWSSEI_ShippingRequest"/>
22        <variable messageType="ns6:CheckLocalAvailability1Request" name="
              CheckLocalAvailability1Request"/>
          <variable messageType="ns6:CheckLocalAvailability1Response" name="
              CheckLocalAvailability1Response"/>
24        <variable messageType="ns6:ReservationRequest" name="ReservationRequest"/>
          <variable messageType="ns6:ReservationResponse" name="ReservationResponse"
              />
26        <variable messageType="ns6:UnReservationRequest" name="
              UnReservationRequest"/>
          <variable messageType="ns6:UnReservationResponse" name="
              UnReservationResponse"/>
28        <variable messageType="ns6:ReserveItem1Fault" name="FaultVariable"/>
          <variable messageType="ns2:ExternalProblemMSG" name="ExternalProblemMSG"/>
30        <variable messageType="ns5:ExternalProblemManagementMSG" name="
              ExternalProblemManagementMSG"/>
          <variable messageType="ns1:ExternalProblemMSG-Supplier" name="
              ExternalProblemMSG-Supplier"/>
32    </variables>
      <correlationSets>
34        <correlationSet name="CS2" properties="ns1:AAACorrelation"/>
      </correlationSets>
36    <sequence>
          <receive createInstance="yes" name="Rec_from_Shop" operation="
              checkAvailable" partnerLink="Warehouse-WH2Shop" portType="
              ns1:WarehouseServer" variable="WarehouseWSSEI_checkAvailable">
38            <correlations>
                  <correlation initiate="yes" set="CS2"/>
40            </correlations>
          </receive>
42        <assign>
              <copy>
44                <from part="parameters" query="/ns4:checkAvailReserve/String_1"
                      variable="WarehouseWSSEI_checkAvailable"/>
                  <to part="parameters" variable="CheckLocalAvailability1Request"/>
46            </copy>
              <copy>
48                <from part="PID" variable="WarehouseWSSEI_checkAvailable"/>
```

```
            <to part="PID" variable="CheckLocalAvailability1Request"/>
50        </copy>
        </assign>
52      <invoke inputVariable="CheckLocalAvailability1Request" name="
            Inv_Check_Local_Availability_on_WH_LocalService" operation="
            CheckLocalAvailability1" outputVariable="
            CheckLocalAvailability1Response" partnerLink="WH-WH2LocalWHService"
            portType="ns6:LocalWHPT"/>
        <switch name="Switch-LocalSupplierInvocation">
54        <case condition="true()">
            <sequence>
56            <assign>
                <copy>
58                <from part="Availability" variable="
                      CheckLocalAvailability1Response"/>
                  <to part="Availability" variable="WHAnswerMSG"/>
60              </copy>
              </assign>
62          </sequence>
          </case>
64        <otherwise>
            <sequence>
66            <invoke inputVariable="Request1" name="
                  Inv_Request_to_localSupplier" operation="asyncOpLocalSupplier
                  " partnerLink="Warehouse-WH2LocalSupplier" portType="
                  ns3:LocalSupplierServer">
                <correlations>
68                <correlation pattern="out" set="CS2"/>
                </correlations>
70            </invoke>
              <receive name="Rec_Answer_from_local_Supplier" operation="
                  onResult" partnerLink="Warehouse-WH2LocalSupplier" portType="
                  ns1:WarehouseServerCallback" variable="callbackResponse">
72              <correlations>
                  <correlation set="CS2"/>
74              </correlations>
              </receive>
76          </sequence>
          </otherwise>
78      </switch>
        <assign>
```

```
80          <copy>
               <from part="PID" variable="WarehouseWSSEI_checkAvailable"/>
82             <to part="PID" variable="WHAnswerMSG"/>
            </copy>
84          <copy>
               <from part="UnPerishableList" variable="
                   CheckLocalAvailability1Response"/>
86             <to part="unReservedItems" variable="WHAnswerMSG"/>
            </copy>
88          <copy>
               <from part="Availability" variable="CheckLocalAvailability1Response"
                   />
90             <to part="Availability" variable="WHAnswerMSG"/>
            </copy>
92       </assign>
         <invoke inputVariable="WHAnswerMSG" name="Inv_Answer_to_Shop_on_Shop"
             operation="receiveAnswer" partnerLink="Warehouse-WH2Shop" portType="
             ns2:WHcallbackPT">
94          <correlations>
               <correlation initiate="yes" pattern="out" set="CS2"/>
96          </correlations>
         </invoke>
98       <pick>
            <onMessage operation="confirmOrder" partnerLink="Warehouse-WH2Shop"
                portType="ns1:WarehouseServer" variable="
                WarehouseWSSEI_confirmOrder">
100            <correlations>
                  <correlation set="CS2"/>
102            </correlations>
               <sequence>
104               <assign>
                     <copy>
106                     <from part="String_1" variable="WarehouseWSSEI_confirmOrder
                            "/>
                        <to part="parameters" variable="ReservationRequest"/>
108                  </copy>
                     <copy>
110                     <from part="PID" variable="WarehouseWSSEI_confirmOrder"/>
                        <to part="PID" variable="ReservationRequest"/>
112                  </copy>
                     <copy>
```

```
114                        <from part="CustInfo_2" variable="
                               WarehouseWSSEI_confirmOrder"/>
                           <to part="CustInfo" variable="ReservationRequest"/>
116                    </copy>
                   </assign>
118                <scope variableAccessSerializable="no">
                       <faultHandlers>
120                        <catchAll>
                               <sequence>
122                                <assign name="Prepare_MSG_for_Shop_and_Supplier">
                                       <copy>
124                                        <from part="PID" variable="
                                               WarehouseWSSEI_checkAvailable"/>
                                           <to part="PID" variable="ExternalProblemMSG"/>
126                                    </copy>
                                       <copy>
128                                        <from expression="'general problem'"/>
                                           <to part="ProblemCode" variable="
                                               ExternalProblemMSG"/>
130                                    </copy>
                                       <copy>
132                                        <from part="PID" variable="
                                               WarehouseWSSEI_checkAvailable"/>
                                           <to part="PID" variable="
                                               ExternalProblemManagementMSG"/>
134                                    </copy>
                                       <copy>
136                                        <from expression="'general problems'"/>
                                           <to part="ProblemCode" variable="
                                               ExternalProblemManagementMSG"/>
138                                    </copy>
                                   </assign>
140                                <invoke inputVariable="ExternalProblemMSG" name="
                                       Inv_Suspension_on_Shop" operation="
                                       ExternalProblemsManagement" partnerLink="
                                       Warehouse-WH2Shop" portType="ns2:WHcallbackPT"/>
                                   <invoke inputVariable="ExternalProblemManagementMSG"
                                       name="Inv_Suspension_on_Supplier" operation="
                                       ExternalProblemsManagement" partnerLink="
                                       Warehouse2RealSupplier" portType="
                                       ns5:RealSupplierServer">
```

```
142                     <correlations>
                            <correlation pattern="out" set="CS2"/>
144                     </correlations>
                    </invoke>
146                 <ext:suspend/>
                </sequence>
148             </catchAll>
            </faultHandlers>
150         <eventHandlers>
                <onMessage operation="ExternalProblemsManagement"
                    partnerLink="Warehouse2RealSupplier" portType="
                    ns1:SupplierCallBackPT" variable="ExternalProblemMSG-
                    Supplier">
152                 <correlations>
                        <correlation initiate="yes" set="CS2"/>
154                 </correlations>
                    <ext:suspend/>
156             </onMessage>
            </eventHandlers>
158         <invoke inputVariable="ReservationRequest" name="
                Inv_Reservation_on_LocalWHService" operation="Reservation"
                 outputVariable="ReservationResponse" partnerLink="WH-
                WH2LocalWHService" portType="ns6:LocalWHPT"/>
        </scope>
160     <assign>
            <copy>
162             <from part="PID" variable="WarehouseWSSEI_checkAvailable"/>
                <to part="PID" variable="SupplierWSSEI_ShippingRequest"/>
164         </copy>
            <copy>
166             <from part="parameters" query="/ns4:checkAvailReserve/
                    String_1" variable="WarehouseWSSEI_checkAvailable"/>
                <to part="ShippingData" variable="
                    SupplierWSSEI_ShippingRequest"/>
168         </copy>
        </assign>
170     <invoke inputVariable="SupplierWSSEI_ShippingRequest" name="
            Inv_ShippingRequest_on_Supplier" operation="ArrangeShipping"
            partnerLink="Warehouse2RealSupplier" portType="
            ns5:RealSupplierServer">
            <correlations>
```

```
172                         <correlation initiate="yes" pattern="out" set="CS2"/>
                        </correlations>
174                     </invoke>
                    </sequence>
176             </onMessage>
                <onMessage operation="unReserve" partnerLink="Warehouse-WH2Shop"
                    portType="ns1:WarehouseServer" variable="WarehouseWSSEI_unReserve">
178                 <correlations>
                        <correlation set="CS2"/>
180                 </correlations>
                    <sequence>
182                     <assign>
                            <copy>
184                             <from part="PID" variable="WarehouseWSSEI_checkAvailable"/>
                                <to part="PID" variable="UnReservationRequest"/>
186                         </copy>
                            <copy>
188                             <from part="UnPerishableList" variable="
                                    CheckLocalAvailability1Response"/>
                                <to part="parameters" variable="UnReservationRequest"/>
190                         </copy>
                        </assign>
192                     <invoke inputVariable="UnReservationRequest" name="
                            Inv_Unreserve_on_LocalWHService" operation="UnReservation"
                            outputVariable="UnReservationResponse" partnerLink="WH-
                            WH2LocalWHService" portType="ns6:LocalWHPT"/>
                    </sequence>
194             </onMessage>
            </pick>
196     </sequence>
    </process>
```

Listing A.3: WAREHOUSE BPEL service

```
1 <?xml version="1.0" encoding="UTF-8"?>
  <!--
3 BPEL Process Definition
  Edited using ActiveBPEL(tm) Designer Version 2.0.0 (http://www.active-endpoints.
      com)
5 -->
```

```
<process name="LocalSupplier" suppressJoinFailure="yes" targetNamespace="http://
    LocalSupplier" xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process
    /" xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
    xmlns:ns1="urn:/LocalSupplierServer/" xmlns:ns2="urn:/WarehouseServer/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
7   <partnerLinks>
      <partnerLink myRole="LocalSupplierProvider" name="LocalSupplier-
          LocalSupplier2Warehouse" partnerLinkType="ns1:LocalSupplier-
          LocalSupplier2Warehouse" partnerRole="WarehouseCallBack"/>
9   </partnerLinks>
    <variables>
11    <variable messageType="ns1:Request" name="Request"/>
      <variable messageType="ns2:callbackResponse" name="callbackResponse"/>
13  </variables>
    <correlationSets>
15    <correlationSet name="CS1" properties="ns1:LocalSupplier-Correlations"/>
    </correlationSets>
17  <sequence>
      <receive createInstance="yes" name="Rec_WH_request" operation="
          asyncOpLocalSupplier" partnerLink="LocalSupplier-
          LocalSupplier2Warehouse" portType="ns1:LocalSupplierServer" variable="
          Request">
19      <correlations>
          <correlation initiate="yes" set="CS1"/>
21      </correlations>
      </receive>
23    <assign>
        <copy>
25        <from expression="true()"/>
          <to part="callbackData" variable="callbackResponse"/>
27      </copy>
        <copy>
29        <from part="PID" variable="Request"/>
          <to part="PID" variable="callbackResponse"/>
31      </copy>
      </assign>
33    <invoke inputVariable="callbackResponse" name="Inv_WH_response" operation=
          "onResult" partnerLink="LocalSupplier-LocalSupplier2Warehouse"
          portType="ns2:WarehouseServerCallback">
        <correlations>
35        <correlation pattern="out" set="CS1"/>
```

```
                </correlations>
37          </invoke>
        </sequence>
39  </process>
```

Listing A.4: LocalSupplier BPEL service

Abstract

This thesis studies the Model-Based Diagnosis focuses on a set of interacting software components. The main idea is to use Colored Petri nets (CPNs) as a fault model, which presents several advantages for software diagnosis. First, we can handel data by avoiding the infinity of their domain values, the data are represented symbolically according to their status (faulty using red color tokens, correct using black and unknown using star). Second, the transition modes are used to represent correct and faulty executions of activities without explicit representation of faults as internal events. Finally, partially ordered observation is naturally expressed in CPNs operational semantics. The main contribution of this thesis is the reduction of diagnosis problem to an algebraic symbolic inequations system based on the fundamental equation of CPNs. This method allows the diagnosis of looping processes and omits the trajectory calculation, without loosing the diagnosis precision. Based on the notion of functional sub-nets, our method can be easily adapted to a decentralized resolution of the inequations systems so as to diagnose the decentralized systems. Our work is applied to the diagnosis of orchestration of Web services modeled as a set of place bordered colored Petri nets. A model transformation from BPEL constructors to CPNs is given and a case study is detailed.

Résumé

Cette thèse porte sur le diagnostic à base de modèles. Nous focalisons notre intérêt sur le diagnostic d'un ensemble interagissant de composants logiciels. L'originalité de ce travail se situe dans l'utilisation des Réseaux de Petri Colorés (RdPC) comme modèle de faute. L'utilisation des RdPC est originale et avantageuse à plusieurs titres. Premièrement, Les RdPC permettent la représentation des données, dans notre cas ceci nous permet de manipuler les données de manière symbolique même si leur domaine de valeurs est infini (seul le statut des données est représenté par des jetons colorés : rouge pour fautif, noir pour correct et étoile pour inconnu). Deuxièmement, chaque transition en RdPC peut avoir plusieurs modalités de franchissement, nous avons donc défini pour chaque activité deux modalités de transition, fautif et correct, auxquelles on a associé des fonctions de propagation de couleur. Finalement, La sémantique RdPC porte de manière implicite la notion d'ordre partiel des observations. La contribution principale de cette thèse consiste à réduire le problème de diagnostic à la résolution d' un système d'inéquation algébrique en se fondant sur l'équation fondamentale de la dynamique des RdPs. La résolution de ce système d'inéquation permet de calculer le diagnostic sans dépliage de la trajectoire même dans les cas d'itération d'activités et ceci sans perte de la précision du diagnostic. Nous avons également, en se fondant sur la notion de sous-réseaux fonctionnels, proposé une version décentralisée de la résolution du système d'inéquation. La dimension applicative de cette thèse concerne le diagnostic d'orchestration de services Web. Une traduction du langage d'orchestration BPEL en RdPC a été donné ainsi qu'une application détaillée sur un scénario.