VLSI Implementation of Petri Net Models of a Class of Digital Systems

by

Asjad Mumtaz Tahir Khan

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

In

ELECTRICAL ENGINEERING

January, 1992
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
VLSI implementation of Petri net models of a class of digital systems

Khan, Asjad Mumtas Tahir, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1992
VLSI IMPLEMENTATION OF PETRI NET MODELS OF A CLASS OF DIGITAL SYSTEMS

BY

Asjad Mumtaz Tahir Khan

A Thesis Presented to the
FACULTY OF THE COLLEGE OF GRADUATE STUDIES
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAH, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

ELECTRICAL ENGINEERING
JANUARY 1992
KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

This thesis, written by

Asjad Mumtaz Tahir Khan

under the direction of his thesis committee, and approved by all the members, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Thesis Committee

Chairman (Dr. Gerhard F. Beekhoff) 9/26/92

Saidq Sait. M 9/26/92

Co-Chairman (Dr. Saidq. Sait)

Member (Dr. Hammar Bawgahy) 9/2/92

Member (Dr. Khalid H. Biyari) 9/2/92

Dr. Abdallah M. Al-Shehri
Department Chairman

Dr. Ala H. Rabeh
Dean College of Graduate Studies

Date: 9/2/92
Dedicated to

My Parents

and

Family
Acknowledgment

I wish to thank my thesis committee chairman Prof. G. F. Beckhoff for his guidance, co-chairman Prof. Sadiq M. Sait for his encouragement and committee members Prof. Mammar Bettayeb and Prof. Khalid M. Biyari for their support.

Thanks are also due to Dr. Timothy Morgan and the Regents of the University of California at Irvine for licensing P-NUT software and the artists who created VLSI Design Tools. I express my thanks to Dr. M. S. T. Benten for his help with the above softwares and for the computing facilities that he has setup.

I also wish to thank the department chairman, Dr. Abdallah M. Al-Shehri for his commitment to lesser work load for teaching assistants and the chairman Computer Engineering department Dr. M. Y. Osman for allowing me to use his department’s facilities. Acknowledgment is due to King Fahd University of Petroleum and Minerals for providing financial support in the form of teaching assistantship without which this work would not have been possible.

My thanks to the graduate students of the departments of Electrical and Computer engineering and my friends from whom I learned a lot and who made the long work hours pleasant.

Lastly, but not the least, I wish to thank Dr. S. Ghanta for introducing me to Petri nets.
Contents

1 Introduction.................................................. 1

1.1 System Level Synthesis.................................... 3

2 Petri Nets: A Review........................................... 6

2.1 Petri Nets: Introduction and Terminology.................. 6

2.1.1 Origin.................................................. 6

2.1.2 Why Petri nets?.......................................... 7

2.1.3 What are Petri nets?...................................... 9

2.1.4 Interpretation........................................... 16

2.1.5 Modelling Example...................................... 17

2.1.6 Sub-classes of Petri nets................................. 20

2.1.7 Extensions to Petri nets................................. 24

2.1.8 Applications........................................... 30

2.2 Petri Nets: Modelling, Properties and Analysis........... 33

2.2.1 Modelling............................................... 33

2.2.2 Properties.............................................. 37

2.2.3 Analysis Techniques.................................... 43
3 Literature Review 49

3.1 Asynchronous Implementation .......................... 50
3.2 Synchronous Implementation ............................ 55

4 Petri Nets, Communication Protocols and Protocol Controllers 60

4.1 Communication Protocols ............................... 61
4.2 Protocol Specification ................................. 65
  4.2.1 Finite State Machine Model ......................... 68
  4.2.2 Petri Net Model .................................. 81
  4.2.3 Protocol Controllers ............................... 86

5 Implementation of Communication Protocols: A Case Study of Distributed Systems 96

5.1 Frontend .................................................. 97
  5.1.1 Decomposition of Protocol ......................... 97
  5.1.2 FSM from the annotated Petri net .................. 113
  5.1.3 Implementation Aspects ............................ 121

5.2 Backend .................................................. 128
  5.2.1 Mealey Equation Generator(meg) ..................... 133
  5.2.2 eqtott ............................................. 137
  5.2.3 espresso ........................................... 141
  5.2.4 mpla .............................................. 142

5.3 Verification ............................................. 142
5.4 Example ................................................. 144
6 Implementation of Concurrent systems from their Petri Net models

6.1 Background and Definitions ........................................ 153
6.2 Concurrency Patterns in Reachability Graph ...................... 157
6.3 Detection of Concurrent Components .............................. 160
6.4 Complexity of the Procedure ...................................... 165
6.5 Implementation Aspects ............................................ 166
   6.5.1 Detection of n-cube .......................................... 166
   6.5.2 Clique number and Chromatic number ....................... 168
6.6 Special Case ..................................................... 177

7 Conclusions and Future Work ...................................... 181

Bibliography ............................................................. 183

Vita ............................................................................ 201
List of Figures

2.1 Representation of places and transitions and tokens. .................. 10
2.2 A Petri net's structural component ........................................ 10
2.3 The complete Petri net with marking ...................................... 11
2.4 (a) The initial Marking (b) Transition $t_1$ fires (c) Transition $t_2$ fires
    , same as (a) ................................................................. 12
2.5 (a) Input and output places of a transition (b) Input and output
    transitions of a place ................................................... 14
2.6 Transforming a generalised Petri net into an ordinary Petri net ...... 15
2.7 (a) A Petri net with Self Loop (b) Self Loop transformed into pure
    Petri net ................................................................. 16
2.8 A Petri net model of a counter .............................................. 18
2.9 The markings of Petri net model of counter ............................. 18
2.10 Petri net showing conflict .................................................. 20
2.11 Petri net showing concurrency ............................................ 21
2.12 Marked Graphs ................................................................... 21
2.13 Petri net with Confusion ..................................................... 22
2.14 Free Choice nets ............................................................... 23
2.15 Extended Free Choice nets .................................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.16</td>
<td>Assymetric Free Choice nets</td>
<td>24</td>
</tr>
<tr>
<td>2.17</td>
<td>Ordinary Petri net</td>
<td>24</td>
</tr>
<tr>
<td>2.18</td>
<td>Relationship between the subclasses</td>
<td>25</td>
</tr>
<tr>
<td>2.19</td>
<td>Examples of different subclasses</td>
<td>26</td>
</tr>
<tr>
<td>2.20</td>
<td>(a) Exclusive OR transition (b) Switch transition</td>
<td>27</td>
</tr>
<tr>
<td>2.21</td>
<td>Petri net with Inhibitor arc</td>
<td>28</td>
</tr>
<tr>
<td>2.22</td>
<td>Nondeterminism in Concurrency structure</td>
<td>35</td>
</tr>
<tr>
<td>2.23</td>
<td>Nondeterminism in Conflict structure</td>
<td>35</td>
</tr>
<tr>
<td>2.24</td>
<td>Fork(t&lt;sub&gt;o&lt;/sub&gt;) and Join(t&lt;sub&gt;1&lt;/sub&gt;) Structures</td>
<td>35</td>
</tr>
<tr>
<td>2.25</td>
<td>Petri net Model of FSM</td>
<td>38</td>
</tr>
<tr>
<td>2.26</td>
<td>Petri net Model with Deadlock. Transition t&lt;sub&gt;4&lt;/sub&gt; is dead</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>The OSI model (courtesy <em>Computer Networks by Andrew S. Tannenbaum</em>)</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>The flow of information in OSI model(courtesy <em>Computer Networks by Andrew S. Tannenbaum</em>)</td>
<td>64</td>
</tr>
<tr>
<td>4.3</td>
<td>The Service Specification</td>
<td>65</td>
</tr>
<tr>
<td>4.4</td>
<td>The Protocol Specification</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>The Layer N in OSI model</td>
<td>69</td>
</tr>
<tr>
<td>4.6</td>
<td>The layer N FSM model with interface information</td>
<td>70</td>
</tr>
<tr>
<td>4.7</td>
<td>The layer N FSM model without interface information</td>
<td>70</td>
</tr>
<tr>
<td>4.8</td>
<td>The layer N Global FSM model</td>
<td>72</td>
</tr>
<tr>
<td>4.9</td>
<td>The pseudo-language description of PAR Protocol(courtesy <em>Computer Networks by Andrew S. Tannenbaum</em>)</td>
<td>73</td>
</tr>
<tr>
<td>4.10</td>
<td>The service primitives of the Data Link layer</td>
<td>74</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.11</td>
<td>The virtual and the real connection between network layer peers</td>
<td>76</td>
</tr>
<tr>
<td>4.12</td>
<td>The FSM model of Sender process of PAR protocol</td>
<td>77</td>
</tr>
<tr>
<td>4.13</td>
<td>The FSM model of Receiver process of PAR protocol</td>
<td>78</td>
</tr>
<tr>
<td>4.14</td>
<td>The FSM model of Channel of PAR protocol</td>
<td>78</td>
</tr>
<tr>
<td>4.15</td>
<td>The global model of PAR protocol</td>
<td>80</td>
</tr>
<tr>
<td>4.16</td>
<td>The global model using a simple interconnection mechanism</td>
<td>82</td>
</tr>
<tr>
<td>4.17</td>
<td>The global model obtained using shared place mechanism</td>
<td>84</td>
</tr>
<tr>
<td>4.18</td>
<td>The global model obtained using merged transition mechanism</td>
<td>90</td>
</tr>
<tr>
<td>4.19</td>
<td>The global model obtained using request-acknowledge mechanism</td>
<td>91</td>
</tr>
<tr>
<td>4.20</td>
<td>The Petri net model of the protocol PAR</td>
<td>92</td>
</tr>
<tr>
<td>4.21</td>
<td>The reachability graph of Petri net model of the protocol PAR</td>
<td>93</td>
</tr>
<tr>
<td>4.22</td>
<td>The architecture of PROVE</td>
<td>94</td>
</tr>
<tr>
<td>4.23</td>
<td>The architecture of Protocol Engine</td>
<td>95</td>
</tr>
<tr>
<td>5.1</td>
<td>Deadlock: The interlocking of processes</td>
<td>99</td>
</tr>
<tr>
<td>5.3</td>
<td>The reduction RSN-4T reduces a shared place to a merged transition</td>
<td>101</td>
</tr>
<tr>
<td>5.4</td>
<td>Illustration of transformations in procedure Break_Entity</td>
<td>107</td>
</tr>
<tr>
<td>5.5</td>
<td>Application of procedure Break_Entities on the Petri net model of PAR</td>
<td>108</td>
</tr>
<tr>
<td>5.6</td>
<td>Application of procedure of Souissi on the modified Petri net model</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>of PAR without loss bieng explicitly modelled</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>Integrated interface-protocol model</td>
<td>116</td>
</tr>
<tr>
<td>5.8</td>
<td>Partitioning of the protocol model</td>
<td>118</td>
</tr>
<tr>
<td>5.9</td>
<td>The FSMs obtained for the sender processes of the protocol PAR</td>
<td>121</td>
</tr>
<tr>
<td>5.10</td>
<td>The subset of P-NUT tool</td>
<td>123</td>
</tr>
</tbody>
</table>
5.11 (a) The textual Petri net model of PAR protocol with entities demarcated (b) The annotated Petri net model of the sender process of PAR obtained by the decomposition procedure .......... 129

5.12 The reachability graph for the sender process of PAR obtained from the procedure for FSM generation for PAR .............. 130

5.13 The sequence of tools used for FSM Silicon Compiler ............ 131

5.14 The sequence of tools used for simulation ......................... 132

5.15 The input meg specification for sender process of PAR protocol . 137

5.16 The output of meg for sender process of PAR protocol in form of truth table ................................................. 138

5.17 The output of meg for sender process of PAR protocol in form of equations .................................................. 139

5.18 The output of eqntott for sender process of PAR protocol .... 140

5.19 The output of espresso for sender process of PAR protocol .... 141

5.20 The layout of the controller of sender process of PAR produced by mpla ...................................................... 143

5.21 A portion of the output of mextra showing the extracted FSM for sender process of PAR protocol .......................... 145

5.22 The timing diagram for sender process of PAR protocol ........ 146

5.23 The textual form of Petri net models of sender and receiver of Alternating bit protocol with entities demarcated .......... 147

5.24 The textual form of annotated Petri net model of sender of Alternating bit protocol ........................................... 148
5.25 The meg input for Petri net model of sender of Alternating bit
protocol ................................................ 150

5.26 The layout of the sender of Petri net model of sender of Alternating
bit protocol. Scalable cmos technology is used with cis style. . 151

6.1 (a) A Petri net with concurrency (b) the reachability graph for the
Petri net .................................................... 156

6.2 2-cube structure required for reflexivity of concurrency relation, co 157

6.3 A 3-cube structure for 3 concurrent transitions ....................... 158

6.4 A 4-cube structure for four concurrent transitions ................... 159

6.5 Application of Breadth First Technique on a 3-cube structure .. 168

6.6 (a) $S_1||S_2$ (b) $S_1S_2||V_1V_2$ and (c) $S_1S_2S_3||V_1V_2V_3$ .... 172

6.7 Illustration of heuristic for detecting sequences (a) The concurrency
pattern (b) table showing the transitions, their predecessors and
parallel transitions and (c) sequences formed ....................... 175

6.8 (a) The Petri net model (b) the reachability graph (c) the CoG
graph with the partite components shown ......................... 178

6.9 (a) The Petri net model (b) the reachability graph and (c) the
transitions not appearing in concurrency graph .................... 180
List of Tables

2.1 Markings of the Petri net interpreted as count .................. 19
2.2 Sequence of transition firing ................................. 19
Abstract

Name: Asjad Mumtaz Tahir Khan

Title: VLSI Implementation of Petri Net Models of a Class of Digital Systems

Major Field: Electrical Engineering

Date of Degree: January 1992

Petri nets are a graphical modelling tool. They are well suited for modelling concurrent, asynchronous and non-deterministic systems. Communication protocol controllers are an example of concurrent/distributed systems. Petri nets have been used in their modelling and analysis. Currently controllers are implemented in software and to accomplish this Petri net compilers exist. However, increasing link speeds make hardware implementation desirable. In this work, automated hardware implementation of concurrent systems from their Petri net models is investigated. As a case study, protocol controllers are considered and a procedure for their automated implementation is presented. The general problem of implementing concurrent systems is also considered and a systematic procedure of exponential complexity for automated implementation is proposed. It is shown that protocol controllers constitute a special case of the general problem and that their implementation can be done in polynomial time.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

January 1992
خلاصة الرسالة

عنوان الرسالة: تشفير نماذج شبكات بتري لفئة من الأنظمة الرقمية باستخدام الحوافز التكاملية (VLSI)

اسم الطالب: أسعد ممتاز طاهر خان
التخصص: المنحسة الكهربائية

شبكات بتري هي عبارة عن وسيلة تشفير صورية. وتشكل هذه الشبكات أداة مناسبة لتمثيل الأنظمة التوافقية الموزعة والأنظمة غير المتزامنة، وكذلك الأنظمة غير اليوتيريدية. تشكل محطات بروتوكولات الاتصالات مثالًا على الأنظمة التوافقية الموزعة. وقد استعملت شبكات بتري لتمثيل وتحليل هذه الأنظمة.

في الوقت الحالي يتم تمثيل المحطات باستخدام شبكات بتري، ثم تترجمها باستعمال الترجمات الآلية إلى برامج تقوم بعمل المحطات. ولكن تبقى عملية تسريع هذه المحطات باستعمال الدوائر الإلكترونية أمرًا مرجوًا فيه.

هذا البحث يشكل محاولة لبناء دوائر إلكترونية، ويشكل آلية الأنظمة التوافقية الموزعة، من نمط شبكات بتري التي تمثلها. كما يقدم مثال دراسي لمحلل بروتوكولات، حيث يعرض طريقة لبناء الآلة بهذه الدوائر.

يقدم هذا البحث طريقة مهنية لبناء الدوائر الإلكترونية للأنظمة التوافقية الموزعة بشكل أسرع. هذه الطريقة ذات تعقيد زمني يتبع شكلاً أساً، كما يقدم أثباتًا على إمكانية تمثيل محطات بروتوكولات، التي تشكل جزءًا من المشكلة الأصلية، ومثل التطبيقات المتعددة الحقول.

جامعة الملك فهد للبترول والمعادن
الظهران - المملكة العربية السعودية
كانون الثاني 1992م
Chapter 1

Introduction

In its infancy, computer software was written at a low level, either directly in binary or using mnemonics for the assembly language instructions and then converted into hexadecimal. However as the size and complexity grew it became error prone and tedious and in cases impossible. This compelled the computer science community to look for a way out from the bottleneck. The concept of high level languages was put forward to overcome this. The programmer was provided a language to put his ideas in. This language was far removed from the assembly language of the computer on which the program was to be executed. It was concise, abstract (with respect to the hardware) and procedural. The programmer now needn’t know about the assembly language of the computer and its hardware. A software, called Compiler, was to do the job of translating this high level description into the assembly language. With this came portability of software, pushing the implementation details to the compiler and the field of software has grown leaps and bounds ever since.
The progress in hardware has been analogous to that of software. Initially the systems were small, just gates, so they could be designed at transistor level. These were then used in the design of simple systems as discrete components. Then came MSI and LSI which used gates as units for designing combinational circuits such as adders etc. Then came VLSI and ROM’s, PLA’s were produced. *High Level Synthesis* aims to emulate for hardware what high level languages did to software. It separates the hardware design process into a frontend and a backend. The design is specified at a high level of abstraction, behavioral or algorithmic level using procedural languages (e.g. Hardware description Languages like VHDL or even Programming languages like Ada or Pascal). Tools can be provided at this level of specification to verify the design. A design-verification cycle can be run at this level to get correct design. Once the design is correct it can be given to an *automation system* which translates this into a structural description of the system. If the output of the automation system is the mask geometry required for fabrication then the automation system is termed a *Silicon Compiler*. High level synthesis has the advantages of fewer errors due to the use of verification tools at a high level, a shorter design cycle due to automation and hence the ability to search the design space [FPC90]. Moreover it brings the IC technology to people in other disciplines who can explore the trade offs in design.

Automation systems for lower levels have been developed and gained acceptance in industry e.g. logic synthesis, state machine synthesis, RTL synthesis etc.. A number of high level synthesis design automation (DA) systems have also been developed. These include *CMUDA* [DPST81] which took design specification in
ISPS and gave a CMOS standard cell or TTL implementation, MIMIOLA [Kro88] from the University of Kiel and CADDY/DSL at University of Karlsruhe [CR80].

Although progress has been made in the design automation of conventional digital systems the explosive growth of VLSI has triggered the design of complex digital systems and spawned new classes of systems. These include today's complex concurrent, distributed and asynchronous systems. The complexity of these systems rules out the use of low level conventional approach to design. Even the current level of abstraction in high level synthesis (algorithmic level) needs to be raised to accommodate these new classes of systems.

1.1 System Level Synthesis

A system level [FPC90] input specification to the high level DA system seems to be a viable alternative. This specification could be in form of a language which supports distributed, concurrent and asynchronous systems. The DA system will, as a first step, partition the specification so that it could be implemented by multiple processors in parallel or pipelined. Then for each of the processes the synthesis system will extract the description of the data path (network of registers, functional units, busses etc.) and the specification of the control part which for synchronous systems can be expressed by a FSM (finite state machine). Finally, if the system is a Silicon Compiler it shall output the mask geometries for fabrication of the entire system using standard cell approach or a fixed architecture.

Along with the various language based contenders, Petri Nets are also a candi-
date for the input specification model. They can model concurrent, distributed
and concurrent systems. They have over the years evolved as a mature modelling
formalism and are backed by established theory. A large pool of tools are available
for automated development, analysis and simulation. They can model a system at
various levels of abstraction, from the system level down to circuit level.

In high level approach to synthesis, acceptable designs (close to the optimal ob-
tained manually) can be obtained by restricting the system to a class of problems.
In areas of distributed/concurrent systems, communication protocols represent
an interesting class. With the proliferation of computers and the advances in
computer communications the area of communication protocols has gained promi-
nence. Huge investments have been made by telecommunication companies in the
area of modelling, verification, testing and automated synthesis. The target for
synthesis efforts has been software. However the inherent limitations of space-
time complexity limit the software implementation. In view of increasing speeds
of communication links a hardware solution seems to be the way out. A high level
synthesis system which takes the protocol specification as input and outputs the
mask geometry for fabrication of the Protocol Processor is an ideal which would
reduce the turn around time for a protocol. The input to such a system can be
specified by using Petri nets. Petri nets are popular in the communication protocol
community for modelling, analysis and simulation purposes and hence their use as
the model for input specification would make use of techniques already in practise.
Work towards the design of a comprehensive Protocol Engineering Workstation is
in progress at the Australia Telecom Research Lab [BWW88]. However, before
this kind of general purpose ‘Silicon Compilation’ DA system can be attempted, techniques have to be developed for efficient automated implementation in hardware from Petri net models.

In this work implementation of Protocol Controllers from their Petri net model will be studied as a case study of concurrent/distributed systems. Further, the general problem for concurrent systems will also be looked into.
Chapter 2

Petri Nets: A Review

2.1 Petri Nets: Introduction and Terminology

2.1.1 Origin

Carl Adam Petri in his doctoral thesis [Pet62] entitled "Kommunikation mit Automaten (Communication with Automata)" presented to the faculty of Mathematics and Physics at the Technical University of Darmstadt, Germany laid the foundation of what has come to be known as Petri net. His work was concerned with the theory of communication between asynchronous components of a computer and the description of causal relationship between events. He presented the viewpoint that the basic phenomenon of communication can be represented by purely combinatorial means and proposed the construction of a net for this purpose. This approach had more practical application in design of information processing machines than provided by the abstract automata approach.
This work came to the attention of research groups in the United States, notable among them are the Information Systems Theory project (IST) at the Applied Data Research Inc. and the Project MAC at the Massachusetts Institute of Technology. The dissertation of Petri was translated to English by the IST project. The work at IST project developed most of the early theory, notation and representation of Petri nets. The current graphic representation originates from them (in the original form the net, as proposed by Petri, did not have the transition as bars). The final report of the project introduced these nets as Condition/Event net [HSSW68]. From 1970–75 the Computation Structures Group at MIT under Project MAC published several theses, reports and memos related to work on Petri nets [Den70].

However after this United States effectively dropped out from Petri net research and most of the work in 80's was done in Europe, especially, Germany and France. In Germany, work was carried out at Bonn's Institut für Informationssystemforschung, Gesellschaft für Mathematik und Datenverarbeitung (GMD) and in France [ADGS80] at Toulouse (Centre of Aerospace Investigations ONERA, Grenoble (National Polytechnical Institut) and Paris (Institute of Programming).

2.1.2 Why Petri nets?

Modelling is widely used to study a phenomenon in almost all fields of study. It gives insight into the real phenomenon without it's physical details. For large computer systems modelling is essential for design and analysis. Growth in VLSI has spawned complex computer systems which include multiprocessor, concurrent
and distributed systems. These involve multiple units communicating with each other and exhibit complex interactions and behavior.

Although Petri's thesis was aimed at communication between asynchronous components of a computer, it tackled features which are common with the above listed type of systems. The features included 'interacting components'. Each component may be a system by itself. Its behavior could be described independently except for well defined interactions with other components. Each component has its own state of existence. This state depends on the past and contains the necessary information to describe the future for given inputs. The components exhibit concurrency i.e., activities of one component of the system occurs in parallel with activities of other components. Moreover, as these components interact synchronization is required between components for transferring information from one component to the other to achieve an overall behavior.

In Petri's work the system under consideration was a computer with its asynchronous components. However, the scenario is similar for modern distributed, concurrent systems like computer communication systems, multiprocessor systems, distributed databases etc. The proliferation of these kinds of systems requires a formal modelling technique for modelling and analysis. The conventional techniques used for modelling sequential systems are inadequate. Petri nets which were developed to handle these situations provide a formalism to model such systems. Apart from this they are capable of modelling conventional sequential systems too. In fact, they can be used to model systems hierarchically at different levels
of abstraction, from system level down to component level.

2.1.3 What are Petri nets?

A Petri net is a graphical modelling tool. This section gives a brief introduction to Petri nets. Four tutorial articles published over the decade and half provide a lucid and complementary introduction [Pet77,Age79,JM82,Mur89]. In addition, two books have been published in English [Pet81,Rei85], others exist in German [Sta80,Rei82]. They have been defined mathematically using the bag theory (an extension of set theory) [Pet81]. However, they can be informally defined as a graph model. An informal introduction to the Petri net model is presented here. The terminology used is defined next. A variety of terms have been used in the literature. The convention followed here is along the lines of [Mur89].

The basic constituents of a Petri net are:

- Places
- Transitions
- Directed arcs
- Tokens

A Petri net is a directed bipartite graph with an initial state called marking. It has two kinds of vertices called Places and Transitions. Places are represented graphically by circles and transitions by vertical bars (sometimes rectangular boxes). Directed arcs exist from a place to a transition and vice-versa but not between
two places or transitions (the graph is bipartite). The places can be marked by tokens represented by black dots. This is shown in Figure 2.1.

Informally, a Petri net model can be said to have two components —

1. Structural – models the static properties of the system

2. Behavioral – models the dynamic properties of the system

The structure is represented by the directed bipartite graph. It has two kinds of vertices called Places and Transitions. Places and transitions are connected by directed arcs. The arcs exist between a place and a transition or vice-versa. It is assumed that there are no isolated places or transitions (which are not connected to any other transition or place respectively) in the net. The Figure 2.2 shows the structural component of a Petri net.
The behavioral component is specified by annotating the places with tokens and the *firing* of transitions. Tokens are represented by black dots in the graph. *Marking* of the net denotes the distribution of tokens in the places comprising the net.

The Petri net model is completely described by its structure and initial marking. The Figure 2.3 shows the Petri net of Figure 2.2 with token added to place $P_1$.

The key concept in describing the behavior of the model is the transition enabling and firing. In fact, it is the only rule to be learnt in Petri net theory. The dynamic behavior of the system is simulated by the transition firing which results in change in marking of the system. The firing rule is given by —

1. A transition $t$, is said to be enabled if each of its input place (places from which a directed arc to the transition $t$ exists) has a token.

2. An enabled transition may fire. Firing is instantaneous, i.e. takes no time.

   As time is a continuous variable, the probability of any two or more events (enabled transitions) happening simultaneously is zero. Hence two or more transitions cannot fire instantaneously.
3. Firing of a transition removes a token from each of its input place and adds a token to each of its output place (place for which a directed arc from the transition t to it exists). The firing is illustrated in Figure 2.4 for the Petri net of Figure 2.3. In Figure 2.4 (a) the transition $t_1$ is enabled as it’s only input place, $P_1$ has a token. When $t_1$ fires the token in $P_1$ is removed and one token is deposited in each of it’s output places, $P_2$ and $P_3$. Figure 2.4 (b) shows the resulting marking. Now transition $t_2$ is enabled as both it’s input places, $P_2$ and $P_3$ have tokens. Firing of $t_2$ removes the tokens of $P_2$ and $P_3$ and deposits a token in the output place $P_1$. This is shown in Figure 2.4 (c).
The resulting marking is the same as the original one. Further firings will repeat the sequence \( t_1t_2 \).

The model introduced, popularly known as Petri nets, is also termed as *Place/Transition net* to differentiate from various mutants that have evolved from Petri nets. A number of variations of Petri nets exist and some of them will be described towards the end of this Chapter.

The Petri net model can be formally specified as:

A Petri net is a 4-tuple, \( PN=(P,T,F,M_0) \)

where,

\( P=(P_1,P_2,\ldots,P_m) \) is a finite set of places

\( T=(T_1,T_2,\ldots,T_n) \) is a finite set of transitions

\( F \subseteq (P \times T) \cup (T \times P) \) is the set of arcs (flow relation)

\( M_0: P \rightarrow N \), where \( N \) is the set of natural numbers

with \( P \cap T=\emptyset \) and \( P \cup T \neq \emptyset \)

The Petri net structure is given by \( N=(P,T,F) \) and

\( PN=(N,M_0) \) denotes a Petri net with marking.

The following notation is used for convenience to describe input and output places/transitions of the Petri net:

\( M(p) \) : number of tokens in place \( p \) in a certain marking

*\( t= ( p \mid (p,t) \in F ) \), the set of input places of \( t \).*

*\( t^* = ( p \mid (t,p) \in F ) \), the set of output places of \( t \).*

*\( p= ( t \mid (t,p) \in F ) \), the set of input transitions of \( p \).*

*\( p^* = \{ t \mid (p,t) \in F \} \), the set of output transitions of \( p \).*
Figure 2.5: (a) Input and output places of a transition (b) Input and output transitions of a place

These are illustrated in Figure 2.5.

Using this notation the following can be defined:

**Self Loop:** \( \ast t \cap \ast t \neq \emptyset \)

**Source Transition of the net:** A transition with no input place, \( \ast t = \emptyset \)

**Source Place of the net:** A place with no input transition, \( \ast p = \emptyset \)

**Sink Transition of the net:** A transition with no output place, \( \ast t = \emptyset \)

**Sink Place of the net:** A place with no output transitions, \( \ast p = \emptyset \)

**Isolated Place:** A place with no input or output transition, \( \ast p = \ast p = \emptyset \)

**Isolated Transition:** A transition with no input or output place, \( \ast t = \ast t = \emptyset \)

The Petri nets described above are called *Ordinary Petri nets.* A variation, *Generalized Petri net,* allows multiple arcs between places/transitions. This can alternately be expressed by assigning weights to the arcs where the weight, \( w \) means existence of \( w \) arcs. Formally, Generalized Petri net is a 5-tuple, the fifth tuple being:

\[ W : F \to \mathcal{N}, \text{ a weight function.} \]
Figure 2.6: Transforming a generalised Petri net into an ordinary Petri net

The firing rule is modified to accommodate this. The transition is enabled only if each of the input place contains $w_i$ tokens, where, $w_i$ is the weight of the arc from a place to the transition. Firing of transition removes $w_i$ tokens from the input place $p_i$ and deposits $w_j$ tokens in the output place $p_j$, where $w_j$ is the weight of the arc from the transition to the place $w_j$. This generalization is just a modelling convenience. A generalized Petri net is equivalent to an ordinary Petri net as an arc of weight $w$ can be represented by $w$ arcs in an ordinary Petri net this is shown in Figure 2.6.

The above Petri net doesn't place any limit on the number of tokens a place can hold and hence is known as infinite capacity nets. Modelling some physical system requires a limit to be placed on the capacity of the places. Such nets are termed finite capacity nets. The transition rule for these is different, called strict transition rule. However, both of them are equivalent in the sense that any finite capacity net with strict transition rule can be transformed into infinite capacity net [Mur89].
Figure 2.7: (a) A Petri net with Self Loop (b) Self Loop transformed into pure Petri net

A net which does not contain a self loop (a place is both an input and output place of a transition) is called a Pure Petri net. A Petri net with self loop can be transformed into a pure Petri net by inserting a place and a transition in the loop as shown in Figure 2.7.

2.1.4 Interpretation

Petri net is a very abstract model. In order to apply it to physical systems certain meaning has to be attached to places, transitions and tokens. This is the interpretation of the net and it depends on the system modelled. Moreover, using different interpretations a system can be modelled at different levels. Hierarchical modelling of large and complex systems is possible with different levels of interpretations.

The condition/event model of systems [GLT79] views the system in terms of conditions and events. The state of the system is defined by the conditions that hold concurrently at a time. The basic unit of change in conditions, which leads a system from one state to another, is an event. For a Petri net model, viewing the
system in terms of condition and event primitives, the input places of a transition can be interpreted as \textit{preconditions}, a transition as an event and the output places as \textit{postconditions}. The conditions that hold are represented by places marked with tokens. Hence the state of the system is defined by the marked places. This view corresponds to the system level in the hierarchy of high level specifications [FPC90]. Other interpretations are possible, depending on the level at which a system is being modelled. For instance, at the algorithmic or instruction set level, the input places can represent input signals, the transitions represent the processor and the output places, the output signal [Mur89,FPC90].

2.1.5 Modelling Example

Petri nets can be used to model a wide variety of systems, from chemical reactions to operating system, compilers and large computer systems. A number of modelling examples are given in the tutorials on Petri nets [Mur89,JM82,Pet77] as well as the books on Petri nets [Rei85,Pet81]. An example from [Roz84] is used to illustrate the significance of interpretations of Petri nets. Figure 2.8 shows the Petri net model of a counter. The initial marking is $M_0 = (1,1,0,0)$. A trace of the execution of the net produces the markings shown in Figure 2.9.

If each of the marking is interpreted as a count (state of the counter) then markings $M_0 — M_3$ denote count of $0 — 3$ and the Petri net models a 2-bit counter as shown in Table 2.1. However if the transitions are interpreted as signals and firing is taken to mean toggling of the signal then the sequence of events is shown in Table 2.2. An asterisk shows the enabled transition. Initially it is assumed that all the signals are low. The firing produces 8 signal which can be taken as counts
Figure 2.8: A Petri net model of a counter.

\[ M_1 = (1,1,0,0) \]
\[ t_1 \]
\[ M_2 = (0,0,1,1) \]
\[ t_3 \]
\[ M_3 = (0,1,0,1) \]
\[ t_2 \]
\[ M_4 = (1,0,1,0) \]
\[ t_3 \]
\[ M_5 = (1,1,0,0) \]

Figure 2.9: The markings of Petri net model of counter
<table>
<thead>
<tr>
<th>Marking</th>
<th>State Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1 = (1,1,0,0)$</td>
<td>00</td>
</tr>
<tr>
<td>$M_2 = (0,0,1,1)$</td>
<td>01</td>
</tr>
<tr>
<td>$M_3 = (0,1,0,1)$</td>
<td>10</td>
</tr>
<tr>
<td>$M_4 = (1,0,1,0)$</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 2.1: Markings of the Petri net interpreted as count

![Diagram](image)

Table 2.2: Sequence of transition firing
and hence the Petri net models a 3-bit counter.

2.1.6 Sub-classes of Petri nets

This section discusses the sub-classes of Ordinary Petri nets. As the generalized Petri nets are equivalent to the ordinary Petri nets, this covers them too.

State Machine (SM) It is an ordinary Petri net such that each transition $t$ has exactly one input place and one output place, i.e.

$$^*t = t^* = 1 \forall t \in T$$

This class of Petri nets can model conflicts (also known as choice or decision). This is shown in Figure 2.10 where either transition $t_1$ or $t_2$ fires. This subclass doesn't allow concurrency which requires more than one output place for a transition. This is depicted in Figure 2.11. This subclass can model FSM's which model decision or choice but do not allow concurrency.

Marked Graphs (MG) It is an ordinary Petri net such that each place has exactly one input transition and one output transition, i.e.

$$^*p = p^* = 1 \forall p \in P$$
This class is a dual of the State machine (SM). They do not allow conflicts but can model concurrency. This is shown in Figure 2.12. When two or more transitions are enabled concurrently both can fire, however the firing is not simultaneous. The firing can occur with no prefixed order.

**Free Choice Nets (FCN)** It is an ordinary Petri net such that every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition.

\[ \forall p \in P, |p^*| \leq 1 \text{ or } {}^*\{p^*\} = \{p\} \]

or equivalently,

\[ \forall p_1, p_2 \in P, p_1^* \cap p_2^* \neq \emptyset \implies |p_1^*| = |p_2^*| = 1 \]
Figure 2.13: Petri net with Confusion

These are a combination of the SM and MG classes as they allow conflicts and concurrency. However, they do not allow both to occur together i.e. confusion. The name free choice comes from the fact that there is no confusion and the choice on which transition to fire for transition with conflict is free. In Figure 2.13 $t_1$, $t_2$ and $t_3$ are enabled but $t_1$ and $t_3$ are in conflict and $t_2$ and $t_3$ are in conflict. Moreover, $t_1$ and $t_2$ are concurrent. Such a situation is called a symmetric confusion. By restricting the sharing of transitions to two places such that if two places share a transition then these do not have any other output transitions, no confusion is allowed to occur. This is exhibited in Figure 2.14.

Extended Free Choice (EFCN) It is an ordinary Petri net such that it allows symmetric confusions to occur by restricting the places which share output transitions to have the same set of output transitions i.e.

$$\forall p_1, p_2 \in P, p_1^\ast \cap p_2^\ast \neq \emptyset \implies p_1^\ast = p_2^\ast$$

This ensures that only one transition from the set will fire. An extended free choice net is shown in Figure 2.15 where $t_1$ and $t_2$ are in conflict as well as
are concurrent. However no asymmetric confusion is allowed. These can be transformed into FC structure [Mur89].

Asymmetric FCN (AFCN) or Simple net (SN) It is an ordinary Petri net such that it allows occurrence of asymmetric confusions only, i.e.

$$\forall p_1, p_2 \in P, p_1^* \cap p_2^* \neq \emptyset \implies p_1^* \subseteq p_2^* \text{ or } p_1^* \supseteq p_2^*$$

Figure 2.16 shows a asymmetric confusion.

Figure 2.15: Extended Free Choice nets
Ordinary Petri net (OPN) The ordinary Petri nets put no restrictions on occurrence of confusion. A confusion structure that is not asymmetric is shown in Figure 2.17.

The relationship between the subclasses is summarized in the Venn diagram shown in Figure 2.18 and Figure 2.19 shows the different subclasses [Mur89].

2.1.7 Extensions to Petri nets

As people started using Petri nets to model real systems a number of modifications and additions were proposed. However, most of these just represented convenience
Figure 2.18: Relationship between the subclasses
<table>
<thead>
<tr>
<th>SM, MG</th>
<th>EFC, FC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t₁</td>
</tr>
<tr>
<td></td>
<td>t₂</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SM, MG</th>
<th>AC, EFC</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>t₂</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FC, SM, MG</th>
<th>PN, AC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>t₁</td>
</tr>
<tr>
<td></td>
<td>t₂</td>
</tr>
</tbody>
</table>

Figure 2.19: Examples of different subclasses
in modelling a particular type of application rather than any extension to the modelling power of Petri nets. These included Exclusive OR transition in addition to the usual AND transitions [Noe71], Switch transitions [Bae73] which had a special input, called switch input, and two outputs, one labelled empty and the other full. When the transition fires, the token goes to either the place labelled empty or full depending on the absence or presence of a token in the switch input place. This is illustrated in Figure 2.20.

The only real extension to Petri nets came as a result of study of modelling power of Petri nets. It was shown that Petri nets couldn't model a Turing Machine as it lacked the ability of zero testing. This led to introduction of Inhibitor arcs which
allowed the all important zero test. The inhibitor arc is shown in Figure 2.21.

If a place is connected as an input to a transition by an inhibitor arc then the transition will be enabled when the input place is empty. This allows testing of the absence of certain condition or the zero test. Petri nets with inhibitor arcs can model Turing Machine. This relates the Petri net to Decision Problem associated with the Turing machine. According to the Church-Turing thesis every solvable decision problem can be transformed into an equivalent Turing machine. This provides the framework for algorithmic computation. Equivalence of Petri nets to Turing machine shows the modelling power of Petri nets. However this means that all the decision problems associated with Turing machines beset the extended Petri nets too.

High Level Petri nets In order to overcome the complex encoding during modelling of certain operations it was proposed to distinguish tokens in some way. This appeared as Colored Petri nets and Predicate/Transition nets [Jen81,Gen81]. Colored Petri nets associate an identification with tokens in form of color. The transition is restricted to check the identification of token for enabling and may change the identification when firing occurs. The Pred-
icate/Transition net considers the identification to be a string rather than a color. The transitions have predicates associated with them to control enabling of transition. The use of high level Petri nets result in compact models as compared to ordinary Petri nets (Place/Transition net) [Jen87,Gen87]. Automatic mapping from compact model to underlying Petri net is also available.

Timed Petri net In the original Petri net model the transition firing is instantaneous and restricted to one at a time. The behavior of the system is given by the complete set of all possible firing sequences. Researchers specializing in stochastic processes proposed considering the firing sequences as being embedded in time rather than ordered in time implying that transition firing actually took some time to occur. This is especially suited to Performance analysis and scheduling problem. Timed Petri nets introduce deterministic delays with the transitions. A number of variation have been proposed:

1. A fixed time is associated with places so that tokens arriving are not available for transition enabling for that period [Sif77].

2. A fixed time is associated with the transition so that tokens that are to be removed are withheld for that time before they appear in output places [Zub80].

3. A variable time is associated with transitions. [MF76] uses a fixed range of firing times for transitions in protocol modelling.

Stochastic Petri nets Here too delays are associated with transitions but they are probabilistic. The markings produced as a result of transition firings is
considered to be a stochastic process. A number of variations exist here too:

1. Most commonly used approach associates an exponentially distributed random variable with each transition as the time required to fire the transition. When a number of transitions are simultaneously enabled, the one with least delay is fired first [Mol82,Nat80].

2. More generalized distributions are used [DTGN84].

3. Generalized Stochastic Petri nets (GSPN) add a transition type that takes zero time to fire, the immediate transition [HS86].

A variety of other types of Petri nets also exist. These include Temporal Petri nets [SL89], Continuous time Petri net [DA87], PIFO net [FM85]. A review of these can be found in [Val81a]. Also from these broad classes hybrid Petri net models have emerged e. g. Stochastic High Level Petri nets [LM88], Colored Stochastic Petri nets [Zen85].

2.1.8 Applications

Petri nets have been used in modelling and analysis in a wide variety of areas. They can be applied to any physical system provided proper interpretations are used. They are especially suited fro asynchronous, concurrent and distributed systems.

Two of the most successful areas of application have been:

1. Communication protocol specification and verification community. They were one of the first ones to experiment with Petri nets. Currently they
are used on a large scale in Australia and Europe especially, France and Germany.

2. Performance modelling and evaluation. Applications under study include parallel and distributed computer architecture [MBC84], software systems such as databases [DC89,NHS83]

New application areas that are emerging are:

1. Software Engineering. Petri nets are being used to model abstract structure of software [MSS89].

2. Distributed and parallel computing.

3. Discrete events, multiprocessor, dataflow and fault tolerant systems [Mur89].


Some other interesting applications are [Mur89]:

- Neural networks
- Digital filters
- Local area networks
- Legal system

So far the application of Petri nets has mostly been in modelling and analysis of systems. Some work has been done in the area of synthesis of digital systems from Petri net models. This includes a recent work [Men91a] on asynchronous implementation. The work in this area is covered in the Chapter on Literature
Review (Chap 3). This work investigates synchronous realization for which no efficient automated methods exist.
2.2 Petri Nets: Modelling, Properties and Analysis

2.2.1 Modelling

Petri nets have been used to model a wide variety of systems (some of the applications were mentioned in the previous section). The tutorials on Petri nets [Mur89, JM82, Pet81, Age79] carry numerous introductory examples. Starting from the Finite State model a number of scenarios are discussed. The condition/event model and the interpretation of the Petri net are crucial in modelling. The system to be modelled has to be fully understood to be faithfully modelled by Petri nets. The basic features of the Petri net model are discussed below:

Asynchronism Petri nets do not have an explicit measure of time. It views the system as an ordering on occurrence of events. They model all the possible sequences and do not take into consideration the time taken by events. This property makes Petri nets especially suited to model asynchronous systems.

Nondeterminism In a Petri net at an instant more than one transition can be enabled. The model provides no deterministic way of choosing which one of these should fire (at a time only one transition can fire). This nondeterminism allows modelling of real life situations where at an instant many things can occur and there is no way to tell which one of these would occur first (e.g. in a circuit if two transistors are coming out of saturation then any one of these could cutoff first) and in modelling asynchronous systems (circuits) where from a number of inputs any of those could change state first. This
is illustrated by Figure 2.22. Nondeterminism occurs in Petri nets in two forms, Conflict and Concurrency.

**Conflict** Another situation exhibiting nondeterminism in a Petri net is a conflict where two or more transitions are enabled by the same input place(s). Firing of one will disable the other. This is exhibited in 2.23. The Petri net provides no mechanism to decide which one of these should fire. This situation is termed as Conflict, Choice or Decision. This models the practical situation where only one of the possibilities could occur.

**Concurrency** A Petri net with more than one transitions enabled represents concurrency. Concurrent events can be modelled by a transition with more than one output places or a fork. A transition with more than one input place is a join. With these two constructs parallel operations can be modelled. An example is given in Figure 2.24. On firing of \( t_0 \) both \( P_1 \) and \( P_2 \) have tokens and transitions \( t_1 \) and \( t_2 \) are enabled. The firing rule limits the firing of transitions to one at a time. So either of the two can fire first (Nondeterminism). Concurrency depicts causal independence between events (in the condition/event model places denote condition and transition are events. \( P_1 \) and \( P_2 \) both have tokens enabling \( t_1 \) and \( t_2 \) denote the fulfilling of two conditions due to which two events can occur concurrently).

**Synchronization** Concurrent systems as well as multiprocess systems need to communicate with each other in order to share information, resources such that the desired global behavior is achieved. One of the processes may have to weight until the desired resource becomes available. This is the Synchroniza-
Figure 2.22: Nondeterminism in Concurrency structure

Figure 2.23: Nondeterminism in Conflict structure

Figure 2.24: Fork($t_0$) and Join($t_1$) Structures
tion problem. A number of Synchronization problems have been described in literature. This includes, Mutual Exclusion problem, Producer/Consumer, Reader/Writer, Dining Philosophers etc. Petri net models of these have been presented and hence, Petri nets can model synchronization.

Finite State Machine FSM is the basic model for sequential systems. The language generated by the machine is Regular. Formally, FSM is defined as:

A FSM is 5-tuple, \( ME=(Q,\Sigma,Y,\delta,\lambda) \) where,

\( Q=\{q_0,q_1,q_2,\ldots,q_n\} \) is a finite set of states with initial state, \( q_0 \)
\( \Sigma=\{a_1,a_2,\ldots,a_m\} \) is a finite set of input alphabet
\( Y=\{y_1,y_2,\ldots,y_k\} \) is a finite set of output alphabet
\( \delta: Q \times \Sigma \rightarrow Q \) is the next state function mapping the current state and input to next state
\( \lambda: Q \times \Sigma \rightarrow Y \) is the output function mapping the current state and input to output

A FSM is also represented graphically by a directed graph whose nodes are states and labelled arcs between nodes are transitions from one state to other. The arcs are labelled by a pair of input and output symbols. Transition from one state to the other occurs when an input labelling the arc from current state to the other occurs, producing the labelled output. As the transition occurs from current state to one of the many possible depending on the input symbol, FSMs model conflict but at an instant the system can only be in one state, hence it cannot directly model concurrency.
FSMs can be modelled by Petri nets. Each state can be represented by a place with the place corresponding to the initial state marked by a token. Directed labelled arcs emanating from a state of the FSM can be replaced by a directed arc from the place representing the state to a transition and another from the transition to next state. The input/output label can be handled in different ways. [Pet81] proposed assigning a place for each input alphabet and output alphabet as inputs and outputs of the net. Each transition contained an input arc from a place representing the input alphabet for which the corresponding edge in the graph was labelled and an output arc from the transition to the output place which labelled the edge. This is illustrated in the Figure 2.25.

2.2.2 Properties

The properties of Petri nets can be classified into two categories:

1. behavioral – those which depend on the structure and marking of the net.

2. structural – those which depend only on the structure only and are independent of the initial marking.

Behavioral Properties

The behavioral properties have been used to answer questions about the Petri net model and hence about the modelled system. These depend on the initial state (marking) of the system. Some of the important properties are discussed below:

Reachability It is the basic analysis problem for the behavior of the net and many other questions about the net can be stated in terms of reachability.
Figure 2.25: Petri net Model of FSM
Definition 1 A marking $M_n$ is said to be reachable from $M_0$ if there exists a sequence of transition firings that transform $M_0$ to $M_n$. 

Let $M_0$ be the initial marking and the sequence of transition firings be given by

$$M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \ldots \xrightarrow{t_n} M_n$$

then the firing sequence, $\sigma$ is denoted by

$$\sigma = t_1t_2\ldots t_n \models M_0 [\sigma > M_n]$$

The set of all reachable markings from the initial marking, $M_0$ is denoted by $R(N, M_0)$ or $R(M_0)$. The set of firings from $M_0$ is denoted by $L(N, M_0)$ or simply $L(M_0)$. With this notation the reachability problem can be formally defined as

Problem Statement 1 (Reachability) Given a Petri net $PN = (N, M_0)$ and a marking $M_n$, determine whether

$$M_n \in R(M_0)$$

The reachability problem has been shown to be decidable [May84] but it takes exponential time and space to solve in the general case.

Boundedness This property is related to the number of tokens in each place of the net.

Definition 2 A Petri net $PN = (N, M_0)$ is said to be $K$ bounded if the number of tokens in each place for all markings reachable from $M_0$ is less than or equal to $K$. 
A special case is of 1-bounded net. These are called safe nets. This property is important when tokens are interpreted as some resource or when the place represents storage or capacity. As long as a place is not a multiple input or output (i.e. Ordinary Petri nets) it is possible to force the Petri net to be safe [Pet81]. Formally, the boundedness property can be defined as

**Definition 3** A Petri net is \( K \)-bounded if

\[
\forall M \in R(M_0) \ \forall p \in P \ M(p) \leq K
\]

Liveness This property is important for Deadlock detection in various applications e.g. resource allocation applications. A deadlock in a Petri net is a state where no transition is enabled and hence no firing can occur. Liveness is basically absence of deadlock, in the weakest sense [Pet81]. Formally,

**Definition 4** A Petri net \( PN=(N,M_0) \) is said to be live if

\[
\forall M \in R(M_0) \ \forall t \in T \ \exists M', \sigma \ M[\sigma < M']
\]

This property, although ideal, is too expensive to detect and hence a weaker definition allowing different levels of liveness has been proposed [Lau75]. Figure 2.26 shows a Petri net with deadlock.

Conservation For Petri nets with tokens interpreted as modelling resources verification of correctness of model or design may be helped by the fact that in the entire system tokens representing resources are constant.

**Definition 5** A Petri net \( PN=(N,M_0) \) is strictly conservative if

\[
\forall M \in R(M_0), \sum_{p_i \in P} M(p_i) = \sum_{p_i \in P} M_0(p_i)
\]
Figure 2.26: Petri net Model with Deadlock. Transition $t_4$ is dead.
A one-to-one mapping between resources and tokens need not be true always. In such cases conservation with respect to a weighting vector specifying the relationship between tokens and resources is used.

Definition 6 A Petri net $PN=\langle N, M_0 \rangle$ is conservative with respect to a weighting factor $W=(w_1, w_2, \ldots, w_n)$, $u = |P|$, $w_i \geq 0$ if,

$$\forall M \in R(M_0), \sum_{i=1}^{P} w_i M(p_i) = \sum_{i=1}^{P} w_i M_0(p_i)$$

A strictly conservative net is a net conservative with respect to $w=(1,1,\ldots,1)$.

Home State and Reversability A Petri net is reversible if from every reachable marking, $M_0$ is reachable.

Definition 7 A $PN=\langle N, M_0 \rangle$ is reversible if

$$\forall M \in R(M_0), \exists \sigma M[\sigma > M_0]$$

In many cases the system goes back to a state other than the initial state. A state is said to be a home state if it is reachable from all states that are reachable from the initial state, $M_0$.

Definition 8 A marking, $M_h$ of a Petri net $PN=(M_0)$ is said to be a home state if,

$$\forall M \in R(M_0), \exists \sigma M[\sigma > M_h]$$

Other properties have also been defined. These include, Persistence, Synchronous Distance, Fairness, Promptness etc. [Mur89,Sif77].
Structural Properties

These are independent of the initial marking and hold true for all initial markings of the net and basically arise from the structure of the net. The properties include Structural Liveness, Conservativeness, Repetitiveness, Consistency, Structural fairness, Invariants. A description of these can be found in [Mur89]. These are not as well studied as the behavioral properties.

2.2.3 Analysis Techniques

There are two approaches to the analysis of Petri nets. They are linked to the two types of properties, Behavioral and Structural.

Behavioral Analysis

There are three techniques for behavioral analysis of Petri nets:

- Coverability Tree
- State Equations
- Decomposition Techniques

These are explained below briefly –

Coverability Tree For the Petri net PN(N,M₀) a tree representation of the reachable markings can be generated. The nodes of the tree are the markings and the arcs represent the transitions whose firing results in that particular marking. The root of the tree is the initial marking, M₀. The firing of transitions enabled in M₀ will lead to new markings. This is represented by arcs
from root labelled by transitions to new nodes representing new markings. The process can go on infinitely, so to limit it whenever a marking is repeated the tree is pruned at that node. However, for unbounded nets this would still be infinite. A new symbol $\omega$ is introduced with the property that, for every integer $n$,

$$\omega > n, \omega + n = \omega, \omega \geq \omega$$

to represent infinity. Places in a marking which show the tendency of increasing monotonically are labelled with $\omega$. This ensures that the tree is finite [Pet81]. This is an exhaustive method of analysis. A number of properties can be obtained from the coverability tree –

1. If $\omega$ doesn’t appear in any marking then the Petri net is bounded.

2. A Petri net is safe if only 0’s and 1’s appear in the markings, formally,

$$\forall M \in R(M_0) \forall p \in P \ M(p) \leq 1.$$ 

3. A transition is dead if it does not appear as an arc in the tree.

4. A marking $M'$ is reachable from $M''$ if there exists a path from $M'$ to $M''$ in the tree.

However, in general (for unbounded nets) the reachability and liveness problems cannot be solved due to loss in information produced by the introduction of symbol $\omega$. Reachability problem has been shown to be decidable [May84].

For bounded nets the coverability tree is known as Reachability Tree and all the problems can be answered from it as it contains all the markings reachable from $M_0$. The coverability and reachability trees can alternatively
be expressed in graph form.

State Equations The structure of Pure Petri nets (Ordinary Petri nets which are not pure can be converted into pure Petri nets) can be represented by an incidence matrix. The dynamical behavior is studied by setting up state equations involving this matrix.

Definition 9 The incidence matrix $A$ of a Petri net structure, $N$ is defined as

$$A = \left( \overline{a_{ij}} \right)^{n \times m}$$

where,

$$a_{ij}^+ = \begin{cases} 1 & \text{arc exists from transition } i \text{ to place } j \\ 0 & \text{else} \end{cases}$$

$$a_{ij}^- = \begin{cases} 1 & \text{arc exists to transition } i \text{ from place } j \\ 0 & \text{else} \end{cases}$$

For general Petri nets the 1 in the definition of $a_{ij}^+$ and $a_{ij}^-$ is replaced by the weight of the arc, $w$.

The marking can be expressed as $M \times 1$ column vector, $M_k^{m \times 1}$, where each entry denotes the number of tokens in that place. A firing control vector, $U_k$ is a $n \times 1$ column vector with a single entry as 1, the transition to be fired, and others 0. The transition enabled, $i$, is the one for which

$$a_{ij} \leq M(j), j = 1, 2, \ldots, m$$
Then the change in the marking as a result of firing of \( k^{th} \) transition can be represented as

\[
M_k = M_{k-1} + A^T U_k, \quad k = 1, 2, \ldots
\]

Reachability Analysis can be performed using the state equations. To find whether a marking \( M_d \) is reachable from \( M_0 \) by a sequence, \( u_1 u_2 \ldots u_d \), following equations can be set up

\[
M_1 = M_0 + A^T u_1 \\
M_2 = M_1 + A^T u_2 \\
\vdots \\
M_d = M_{d-1} + A^T u_d \\
M_d = M_0 + A^T \sum_{i=1}^{d} u_i \\
\Delta M = A^T \sum_{i=1}^{d} u_i
\]

where, \( \Delta M = M_d - M_0 \). Applying the rules of Linear Algebra [Mur77] gave the necessary condition for reachability. Using this approach various properties can be found, however, it is limited by the fact that the solution has to be a positive integer and that loss of sequencing information of firing sequences occurs.

**Reduction Rules** Transformations which do not change the property under consideration can be applied to the net to reduce it into a smaller one so that other computationally expensive techniques (e.g. Reachability tree) can be used [LFB87].
Structural Analysis

As the structural properties are independent of initial marking they can be evaluated using the incidence matrix only which represent the structure of the net. Moreover, they are not beset with the problem of state explosion associated with the behavioral techniques. The basic tool in structural analysis have been invariants. Invariants of two kinds have been defined, S and T invariants.

T-Invariant A T-invariant is a set of transitions such that a firing sequence involving these brings the Petri net back to it's initial marking. Considering the State equation of a Petri net,

$$\Delta M = A^T \sum_{i=1}^{d} u_i$$

or,

$$\Delta M_x = A^T x$$

$$A^T x = 0$$

where, $x$ contains a 1 for the transitions that fired during the sequence. Formally,

Definition 10 $x$ is a T-invariant iff

$$x \in T, \forall M \in R(M_0), \ x^T M = x^T M_0$$

S-Invariants A S-Invariant is the set of places for which the number of tokens remain constant for all possible markings. It is formally defined in the same way as T-invariants. It is obtained by the solution of

$$A.v = 0$$
Definition 11 \( v \) is a \( S \)-invariant iff

\[ v \in P, \forall M \in R(M_0), \quad v^T M = v^T M_0 \]

\( S \) and \( T \)-invariants play a useful role in finding structural properties of the Petri net. [MR79] deals with application of \( S \) and \( T \) invariants for this purpose.
Chapter 3

Literature Review

In the last two decades Petri nets have been an active area of research. The aim in this work is to implement a given Petri net in hardware. It is assumed that the given Petri net models a real digital system. This means that the model is live and bounded. In this chapter a review of the approaches to hardware synthesis from Petri net model is done. Two distinct implementation strategies are possible

- Asynchronous implementation

- Synchronous implementation

Due to the asynchronous nature of Petri net model, the initial target was an asynchronous implementation. Work on this line was carried out in the Project MAC at MIT and in Italy simultaneously in the early 70's. The basic area of application was design of control structures for digital systems (computers, in particular) where asynchronous implementation was faster than a synchronous one and provided a speed up in the operation of the system. In the next two
sections work done in both asynchronous and synchronous areas of implementation is discussed.

3.1 Asynchronous Implementation

The Computation Structures group at MIT under Project MAC pioneered work on implementation of Petri nets. In the earliest reported work completely speed independent (the introduction of arbitrary delays, both in elements and their interconnections, have no effect upon circuit operation) abstract modular structures for implementing Petri nets were proposed [Pat70]. As a sequel to that Patil [Pat71] presented circuit implementation of the modules for safe Petri nets. The implementation was not completely speed independent (this compromise reduced the complexity of the modules) and assumed that an upper bound can be set on transmission delays of signals. The basic idea behind the implementation was to replace each place and transition by an equivalent circuit module. The modules were based on the Muller model for synthesis of asynchronous schemes [Mul71,MB59]. A token was interpreted as a signal and signals were represented by changes in voltage level of wires. The interpretation was different for transition and place blocks. On wires coming out of transition block, every change in level was interpreted as a token whereas at the input/output place of block a 0–1 change was interpreted as a token. To resolve conflicts arbiters were required.

This group later proposed a Petri net based scheme for description and asynchronous realization of digital systems [Pat72]. The data path was represented by registers, operators with control lines to communicate with the control section
and deciders, interconnected using data links. The control part was modelled by a Petri net with transitions labelled by operators from data part and outcomes of deciders. The communication between control and data parts was by control links which had two signals, ready and acknowledge. The ready signal was sent from the control to the operator to signify the presence of operands. When the operation was complete, the operator responded by making the acknowledge line high. Thus the flow in the data part was not delayed beyond the time required for actions. They presented synthesis rules for a class of live and safe, persistent, labelled Petri nets (which they called, “Simple nets” – each transition has at most one shared input place). The implementation used simple signalling in which any change in signal level on a wire corresponded to an event in the net. Corresponding to this interpretation 8 Petri net structures and their circuit equivalents were proposed. They claimed that application of these eight rules to a control net transforms it into a circuit that could be realized using eight control modules proposed by them. This realization too required arbiters to resolve conflicts. This work was continued by [Fur71] who designed a number of modules to realize arbitrary, safe nets.

This work was used by Grandone and Zerbetto to realize \( \mathcal{L} \)-graphs. They extended labelled Petri nets to allow pipelining of operations and introduced \( \mathcal{L} \)-graphs (related to Slutz's flow graph [Slu68] and Karp & Miller’s Parallel Program Schemata [KM69] ) as graph model of digital systems. It was sought to combine all the aspects of the design viz-a-viz description, analysis and realization through this model. \( \mathcal{L} \)-graphs of any complexity, describing control schemes were translated into single conflict Petri nets. This could then be implemented by a network of
asynchronous modules proposed by [Pat72].

In an approach similar to Patil’s [Pat70] Cavarroc proposed replacing each place of a stable and safe net by a trigger with valve, called David’s cell [CBG74]. The setting of a trigger denoted a token. The interpretation of firing was modified so that the tokens were first put in the output places and then removed from the input places. This interpretation facilitated the hardware implementation by David’s cell. For multiple input and output places, the triggers were made of AND-OR-NOT elements. The triggers were provided with luminous elements to aid visualization of the control process in industrial automation. However the loops in the Petri net were to have at least 3 places for correct implementation.

Breaking from previous approaches, Misunas developed a repertoire of safe and conflict free Petri nets, called P-nets modelling desirable circuit behavior [Mis73]. Circuit modules simulating the operation of these nets were developed using Muller’s Consensus element (C element) [Mul63] and gates, K-circuit (speed independent AND), P-module (SR flip flop), Delay etc. He proposed a twin approach to speed independent design in which either the basic P-nets could be combined to model the desired behavior and then converted to hardware or the circuit could be designed and realized using the speed independent modules and their operation could be verified by the net made of their P-net equivalents. He illustrated his approach by designing a processor for synthesizing music.

A more recent approach [Cou83] to realize Petri nets by asynchronous circuit too
was based on the lines of Patil's approach [Pat71]. In the realization, each place is replaced by a place module and each transition by a transition module. The Petri net can be described, in general case (with loop), by a forward and a backward incidence matrix. The realization process is straightforward and similar to that of a PLA. A matrix of transition-place is laid out. The existence of interconnection is decided by the presence of arcs in the Petri net. The implementation method is valid for general, bounded Petri nets. The basic philosophy is that places which store tokens are realized by up-down counters which keep track of the token count (the counters can count up to $k$, where $k$ is the bound of the net). The transitions primarily do the job of checking the token count in their input places and when the condition for firing is satisfied, they fire, causing the token count to change in the input and output places. The modules operate synchronously internally but their interconnection is asynchronous (similar to Patil's modules which were not completely speed independent in their internal behavior, however at terminals their operation appeared to be so). Handshaking signals were used to ensure correct operation.

Considering the above stated approaches to synthesis it is evident that the criterion for implementation was ease of mapping the Petri net into equivalent circuit form. Either the Petri net components, the places and the transitions, were replaced by their equivalent circuit modules and a scheme for handling asynchronous communication along with the interpretation of token was given or at a higher level, circuit implementation for certain basic Petri net structures was given and the given Petri net was assembled from these. These approaches traded efficiency for ease and reg-
ularity (standard cells) of implementation (though the routing required for VLSI implementation would be complex). Moreover the interpretations used limited the abstraction of details at the time of modelling. The implementations depended on local timing (local to modules) and the timing constraint were the same as those for Huffman asynchronous state machine [Chu86b].

In his study Molnar argued that the Petri nets may not be a good model for synthesis of asynchronous circuits as markings (state) of a Petri net have no direct relationship with state of a asynchronous circuit where a state is defined as vector of signals [Mol85]. He concluded that Petri nets can only model the behavior of asynchronous circuits and can be used only as a useful analysis or synthesis tool.

Building on Molnar's arguments, Tam [Chu86b] proposed Signal Transition Graph, a derivative of Petri net, for describing actions of asynchronous control circuits. Signal transition graphs (STG) describe the operation of a circuit by specifying the precedence relationship between transitions of signal in a circuit. They are basically interpreted marked graphs [CH71]. The transitions are represented by signal transitions, the places are omitted without ambiguity as in marked graphs each place can only have one input and one output transition. The tokens are placed on arcs. However there are some differences, the transitions always occur in pairs, labelled as $x^+$ and $x^-$ representing the change of transition from 0→1 and 1→0 respectively. A scheme for synthesizing asynchronous circuits from STG model was presented [Chu86a]. The reachable states for the STG were generated. Liveness and safeness were proposed as the necessary and sufficient conditions for
which each of these states could be assigned a unique bit vector (state). From this assignment and the reachability tree the logic equations for the signals were determined.

Building on the work done by Tam, Meng et al. proposed a system for automatic synthesis of asynchronous interconnection circuits from their high level specification [MBM89, Men91b]. A four-phase handshake protocol is used as the underlying scheme for realization. The hardware for synthesizing this scheme can be built using Differential Cascode Voltage Switch Logic (DCVSL). The system takes a behavioral high level description of the circuit using deterministic guarded commands [Diij75]. This description is translated into Signal Transition graphs (basically, Free Choice nets). They presented necessary and sufficient conditions on STG for speed independent synthesis (strongly connected, unique bit vector assignment and persistence) and for hazard free implementation (semi-modular STG's). The STG is checked for these conditions and modified to impose weakest semi-modularity to achieve maximum concurrency. Once this is done then the state diagram is obtained from the STG and state assignment is done. Using standard minimization techniques, boolean expressions are obtained for each signal. These equations are fed to CAD tools to be realized using AND, OR and C gates which are implemented in VLSI using DCVSL.

3.2 Synchronous Implementation

Synchronous implementation of Petri nets have also been proposed with PLA as the target architecture. Kwan et al. proposed a strategy for the design of struc-
tured digital systems [KBM77]. The data part was to be implemented by a bit slice \( \mu P \) and memories (ROM and RAM). The control part was modelled by a Petri net and implemented by a PLA. They considered live, safe ordinary Petri nets for implementation. They presented a method for generating the PLA table for a subclass, State Machines (each transition has only one input/output and only one place is marked). Places were assigned outputs and the transitions were labelled with inputs (Moore model). For a SM with \( P \) places the number of states was \( P \) (as only one place was initially marked). Each place (state) could be assigned a unique bit vector of size \( N = \lceil \log (P + 1) \rceil \). The PLA table was generated by considering each pair of places, \( a \) and \( b \). If there was a transition from one to the other an entry in PLA table was created. The present state was the bit vector of \( a \), the input condition was the label assigned to the transition between the two places. Two lines were generated, one for the case when input condition was not true, the next state remained the same and the output associated with the place was asserted. The other when the input condition was satisfied, the next state was state \( b \) and its associated output was made high. If the input condition was a sum of products, each product term contributed to such an entry. When a place had more than one output transition then only one of the conditions labelling them was allowed to occur (conflict free, deterministic) and the occurrence of each of them individually contributed to an entry in the PLA table.

When the Petri net was not a SM they gave a method for decomposing it into SM's. This required that only one place be initially marked, a final place existed which was connected to the initial place and that the Petri net was complete, i.
e. there were no source and sink places. The decomposition method was to cut all those transitions which had more than one input/output place. Each of the connected component was replaced by a macroplace and a macronet was made by connecting the macropaces as in the original net. The macroplace corresponding to the connected component which had the token was marked. Those macroplaces which were not simultaneously marked are combined into a single strongly connected Petri net so as to have a single initial place. The synchronization between the component SM's was achieved by associating information about the marking of the subgraph of the macroplace to the transition to which the macroplace was connected. Each component SM would then be implemented by the method stated previously. The criterion for grouping the macropaces was left to the user and the number of STG's, inputs of STG's, output's of STG's were proposed as measures of partitioning.

Auguin et al. proposed another method for synthesis of live and safe Petri nets [ABA80]. They too interpreted marking as a state of the system and assigned boolean function to the transition to model external inputs along with a always true dummy input, λ. They associated two types of outputs, Moore type with places and Mealy type with transitions. For nets of small sizes they exhaustively generated the marking class for the initial marking and to each assigned a bit vector. Change from one marking to the other occurred as a result of firing of a transition when its input places were marked and the boolean function labelling it was true. Each transition from one marking to the other contributed to two lines in the PLA table, one for the condition being satisfied and the other when it was
not. If in a certain marking more than one transition was enabled they treated all of them occurring at once and generated only one boolean function corresponding to the covering of the conditions for all the transitions. For the unlisted combinations in the PLA table the PLA output was zero. This was to be handled by having a T flip-flop at the output register or a new output which masked the clock of the output register.

They presented a method similar to Kwan’s for decomposing large Petri nets. As a first step, all the arcs of transitions with more than one input or output were cut. This resulted in a set of connected components. A macroplace was associated with each connected component and the original net rebuilt using the macroplaces. This was called the macronet. A choice was made on the connected components to be implemented by a PLA (this was left open, the possibilities suggested were, a partition on the set of input variables, output variables, minimum realization in terms of PLA). A procedure for extracting the macronet components to be implemented by a single PLA was given. The macronet was to be expanded by substituting the connected components instead of the macroplace. The remaining macroplaces were called joining places. For synchronization between the PLA’s, for every input/output transition of a joining place a new variable was generated which stood for the transition being enabled. For any output transition of a joining place, the associated function was ANDed with an extra condition which was true when the place in the connected component representing the joining place enabled the transition. Finally transformations could be applied to joining places to eliminate redundant ones. However, this method of decomposition would fail
if every transition of a Petri net happens to have more than one input/output place.

Konig et al. [KF84] proposed a different approach for decomposition. In their scheme, the given Petri net was decomposed into SM's and each was implemented by a Corex memory structure of fixed size. The optimality criterion was the number of modules and the interconnection between them. They proposed a set of algorithms based on coloring of graph vertices which yielded an approximate solution readily. This solution was further refined until the optimal criterion was achieved.
Chapter 4

Petri Nets, Communication Protocols and Protocol Controllers

This chapter discusses the application of Petri nets in protocol modelling and explains what is meant by a Protocol Controller. Communication protocols are one of the two most successful areas of application of Petri nets, the other being performance evaluation [Mur89]. The protocol specification and verification community were one of the first ones to experiment with Petri net modelling. The Australian and the French telecommunication industries are using this model on a large scale although the United States uses a language based specification. In fact, Petri net models are being studied more in the industry rather than at universities. Petri nets are suitable for fields which have distributed or parallel components. Communication protocols are an instance of this class, software engineering especially distributed software, distributed and parallel processing are others.
Petri nets have been used for specification and verification by the communication protocol community. The protocol in question is modelled by a Petri net and then this model is analyzed for desirable properties. If some error is found in the behavior of the protocol then the model is changed and this cycle is continued till the protocol exhibits desired behavior. Once the protocol is found to be correct it is implemented in software. This work proposes to extend this modelling-analysis cycle to include implementation in hardware so that the entire design cycle, from specification to realization can be done using Petri model. This has the obvious advantage of eliminating translation from one specification to another and the possible errors associated with it. Further a hardware realization has the advantage of higher speed over software. Work on translation of Petri net model to high level programming language exists [1189]. The next sections take a look at protocol modelling and protocol controllers.

4.1 Communication Protocols

The VLSI revolution has resulted in bringing down the size and cost of computers and at the same time has increased their processing power. All this has led to proliferation of computers and with it came the need for computers to share information. A Computer Network is a system of a large number of interconnected computers. After an initial period of haphazard growth, standards have been put forward for networks.

International Standards Organization (ISO) has proposed the Open Systems In-
terconnection (OSI) Reference model [DZ83]. This model provides the network architecture which is divided into seven levels in order to reduce design complexity as well as to allow compatibility between different networks. Figure 4.1 shows the model. Each level is termed a layer and each layer provides a certain service to the layer above it and shields it from the actual implementation details of the service. A layer provides service to the one above it using the service of its lower layer. A layer on one machine communicates with the corresponding layer on the other, virtually. The actual communication is via a chain of lower layers. This is illustrated in Figure 4.2. The implementation of a layer on a machine is called an entity (it could be a software, a process or a hardware, a chip). The entities comprising the corresponding layers on different machines are called Peer Process. The rules and conventions used in conversation between peer processes is termed as the Protocol for that layer. A protocol is based on message exchange, this includes synchronization messages between entities that wish to communicate before they can handle data and the message transfer that occurs during data exchange.

Information is transformed from the sender to receiver via packets with headers as shown in Figure 4.2. The sender’s user process gives the data to the application layer which attaches the application layer header, AH and gives it to the presentation layer. The presentation layer modifies it, attaches a header and gives it to the lower layer. The process is repeated until it reaches the physical layer where the packets are actually transmitted to the receiver. At the receiver various headers are stripped as the message propagates in the reverse direction until it reaches the receiver’s user process. Although the data travels vertically, a layer z in the
Figure 4.1: The OSI model (courtesy Computer Networks by Andrew S. Tanenbaum)
Figure 4.2: The flow of information in OSI model (courtesy Computer Networks by Andrew S. Tanenbaum)
sender assumes that it is sending the data to it's peer $x'$ in the receiver and that the data flow is horizontal.

4.2 Protocol Specification

The protocol specification for a layer involves *service specification*, *interface specification* and the *protocol specification* [BS80]. As described in the previous section, in layered architecture each layer does some work for the upper layer called 'service'. From the point of view of layers above it, the input-output behavior of the layer constitutes a service specification. It is an abstract black box description. This is shown in Figure 4.3. It is based on a set of abstract service primitives which describe the input/output functions e.g. for data link services basic primitives are send, receive etc. These service primitives are executed in a certain order which is given in the specification. The order may depend on constraints due to operations by the same user (local) or other user (global). The execution of service primitives involve exchange of parameter values between the adjacent (vertically) entities.
Figure 4.4: The Protocol Specification

The exact format of how the services are provided is defined by the Interface specification which could be different for different users.

Although interaction occurs between adjacent layers (vertically) the basic concept behind the layered architecture and the design of layers is that each entity assumes that it is dealing with its peer. In the network with physically separated users the protocol layer is distributed and entities local to each user communicate with one another via services of lower layers. The rules for interaction between entities in providing the layer’s service constitutes the layer’s actual protocol. The term communication protocol, as commonly used, refers to this protocol. Figure 4.4 shows a symbolic representation of a protocol. A protocol describes the operation of each entity in the layer in response to user commands, messages from
other peer entities and internally generated actions. The protocol specification is basically a refinement or distributed implementation of the service specification.

The protocol specification techniques broadly fall into three categories –

- Finite state transitional models
- Programming language approach
- Hybrid of transitional and Language approach

The finite state models include Finite State Machines (FSM), directed graphs, Petri nets etc. and the programming language approach includes special languages designed for parallel/distributed systems including Estelle [Jr86], Lotos [Bri86] etc. As the focus here is on hardware implementation the programming language will not be looked into though current efforts on High Level Synthesis from languages may make synthesis feasible from them too. One of the aims of protocol specification is to provide a basis for implementation [BS80]. The implementation was supposed to be done in software. The programming language approach is suitable in this case and a direct implementation is possible. However when transitional models were used these were compiled into an implementation. In this work, a compiler of a different sort, Silicon Compiler, is proposed.

The transitional models are based on the fact that protocols consist of simple processing in response to occurrence of events such as commands from the user, message arrival from lower layer) and internal events (such as timeouts). The general approach in modelling protocols is to consider the protocol as a distributed
system (which it is) [Dia82]. As shown in Figure 4.5, the layer N is assumed to be distributed. Each site can be represented by a process and the overall model can be represented by individual processes with inputs and outputs which communicate or interact with each other. This then corresponds to the model of interprocess communication which has been studied and the model reduces to one of communicating processes. The entire network can be studied with this modular multilevel approach [Dia82]. The other approach to model the entire network is to use top-down technique [Val79] but it is more difficult due to interactions that occur in the distributed systems. In the next subsections the two popular transitional models are investigated using the multilevel approach.

4.2.1 Finite State Machine Model

The FSM model has been widely used for protocol modelling. The basic concept of representing a protocol is that each entity is modelled as a FSM. A FSM is in a certain state at any instant of time. This state is defined as the values of variables associated with the entities. Typically states are chosen to be those instants when the entity is waiting for the next event to happen [Tan87,Dan80]. The channel connecting the sites is also modelled as a FSM to incorporate the characteristics of the medium (for e.g. loss, where, loss means a delay in the arrival of packet or error in the packet). The global FSM can be obtained as a composition of the individual machines. The overall behavior of the protocol is then given by a subset of the state space of the global FSM. Figure 4.6 shows this scheme. It also includes the interface between the user and the Nth layer. The inputs and outputs of each site process can be separated into two categories, input/output with the
Figure 4.5: The Layer N in OSI model
Figure 4.6: The layer N FSM model with interface information

Figure 4.7: The layer N FSM model without interface information

peer process and the input/output with the local user. The local component can
be treated separately as Interface protocol as it could be different for different
users. This leads to the model shown in Figure 4.7.

Using this approach each site processor can be modelled as a FSM. The inputs
of site A FSM are the outputs of the channel FSM and its outputs are inputs to
the channel FSM. The same holds true between FSM of site B and the channel.
However some of the events occurring in a process might be internal to it. These
events occur outside the specification of the protocol but are related to the correct behavior. They in a way imply that not only the protocol but some detail of the process implementing it is also modelled. These internal events occur without any external input and may or may not produce an output. In the FSM model they can be represented by the always true input \( \epsilon \). With these, each process can be defined as a FSM with:

\[
Q = \text{a finite set of states with initial state, } q_0
\]

\[
\Sigma = \text{set of frames or messages that can be received from the receiver including } \epsilon
\]

\[
Y = \text{set of frames or messages that can be sent to the receiver including } \epsilon
\]

\( \delta, \lambda \): output and next state functions describing the protocol

The resulting overall FSM model is shown in Figure 4.8. It is a combination of two feedback connections.

As illustration of this approach an example of a Data Link Layer protocol, PAR, Positive Acknowledgment with Retransmission, from [Tan87] is presented. A pseudo-language high level description of the protocol is given in Figure 4.9 But before discussing it, it would be worthwhile to take a look at the data link layer. It is the 2\(^{nd}\) layer in the hierarchy, just above the physical layer. It provides various services to the network layer, the primary being to transfer data from the network layer on source machine to the one on destination process i.e. act as a reliable bit stream for the network layer. It deals with the construction of a frame, transmission errors and link management issues. The connected oriented service has
Figure 4.8: The layer N Global FSM model
(Protocol 3 (par) allows data to be transmitted in one direction over a
noisy communication channel that garbles and even loses frames.)

const MaxSeq = 1;

type EvType = (FrameArrival, CksmErr, TimeOut);

procedure sender3;
var NextFrameToSend: SequenceNr;
   s: frame;
   buffer: packet;
   event: EvType;
begin
   NextFrameToSend := 0;
   FromNetworkLayer(buffer);
   repeat
      s.info := buffer;
      s.seq := NextFrameToSend;
      ToPhysicalLayer(s);
      StartTimer(s.seq);
      wait(event);
      if event = FrameArrival then
         begin
            FromNetworkLayer(buffer);
            inc(NextFrameToSend)
         end
      until doomsday
   end; {sender3}

procedure receiver3;
var FrameExpected: SequenceNr;
   r,s: frame;
   event: EvType;
begin
   FrameExpected := 0;
   repeat
      wait(event);
      if event = FrameArrival then
         begin
            FromPhysicalLayer(r);
            if r.seq = FrameExpected then
               begin
                  ToNetworkLayer(r.info);
                  inc(FrameExpected)
               end
            end
      until doomsday
   end; {receiver3}

Figure 4.9: The pseudo-language description of PAR Protocol (courtesy Computer Networks by Andrew S. Tannenbaum)
basically three phases:

- establishment of connection between sender and receiver,
- data transmission,
- connection release.

The communication between the Network and Data Link layer is through OSI service primitives – request, indication, response and confirm. Request asks the data Link layer to do something, Indication is used to indicate to network layer of the receiver that something happened. Response is used by receiving network layer to reply to an ‘indication’ and confirm is used by the data link layer to the network layer (sender). This exchange is depicted in Figure 4.10 and describes the interface of Data Link layer with the network layer.
The network layer sends the data as a packet to be transmitted to the receiver network layer. The virtual connection between the peers is implemented by the data link layer using the services of the physical layer. It encapsulates the packets into frames and gives them to the physical layer. The whole process is depicted in Figure 4.11. Now the PAR protocol is presented.

Protocol 1 (PAR) Positive Acknowledgment Retransmission

It is a Simplex Data Link layer protocol for a noisy channel. The high level pseudo-language description is given in Figure 4.9. The sender transmits a frame with a sequence number, either a 0 or a 1. Initially, it transmits frame₀ and waits for the acknowledgment, ack to arrive from the receiver. If no ack arrives within a certain prefixed time or if an error is detected in the arrived ack it assumes that the frame was lost and retransmits. If an ack arrives it transmits frame₁ and waits for its ack to arrive. The receiver on the other hand sits idle initially expecting frame₀ to arrive. If a frame₁ arrives it rejects it and sends an ack, and stays in the same state. When frame₀ arrives it accepts it, sends an ack, and goes to the state waiting for frame₁ to arrive in a similar fashion. The channel can be in four states, frame₀ on way, frame₁ on way, an ack on way and the channel being empty. These can be modelled as FSMs. These are shown in Figures 4.12, 4.13, 4.14. The sender has two states, one waiting for acknowledge for frame₀ and the other for frame₁. The receiver too has two states, one expecting frame₀ and the other expecting frame₁. The channel has four states and is incompletely specified as a number of input combinations will not occur.

The global FSM can be built from the cartesian cross product states of the in-
Figure 4.11: The virtual and the real connection between network layer peers
Figure 4.12: The FSM model of Sender process of PAR protocol
Figure 4.13: The FSM model of Receiver process of PAR protocol

Figure 4.14: The FSM model of Channel of PAR protocol
dividual FSMs. The actual interconnection is of the type shown in Figure 4.8. It has $2 \times 2 \times 4 = 16$ states. However, the actual protocol has only 10 states and is shown in Figure 4.15. The global state is depicted as a 3-tuple, $(S,R,C)$, where a 0/1 for $S$ denotes the frame number Sender has sent, for $R$ it denotes the frame expected and the Channel may have a frame, 0/1 or an acknowledgment, A or it can be empty, . The initial state is chosen as (0,0,0) i.e. sender has sent $frame_0$, is expected by the receiver and the channel has it. Nine kinds of transitions are depicted. Transition 0 results in the channel losing it's contents. Transition 1 consists of channel delivering $frame_0$ to the receiver and the receiver sending an acknowledgment and changing state to expect $frame_1$, (0,1,A). Transition 7 denotes retransmission by the sender in case of loss (0,0,-) to (0,0,0). The acknowledgment sent by the receiver might get lost (0,1,-), transition 0 in which case a retransmission by the sender occurs. When the acknowledgment arrives at the sender, transition 2, the sender sends $frame_1$ and goes to state (1,1,1) waiting for the acknowledgment. The same scenario is repeated for $frame_1$. Transition 3 is similar to 1, 4 to 2, 5 to 6 and 7 to 8. Once the acknowledgment for $frame_1$ arrives, the protocol returns to it's initial state. During normal operation transition numbers 1,2,3 and 4 are repeated in that order delivering two frames per cycle.

FSM model has the disadvantage that the global FSM may contain extraneous states obtained by the cross product of the individual FSM's, which do not belong to the protocol specification. Moreover it is difficult to relate the global states with the behavior of individual processes.
Figure 4.15: The global model of PAR protocol
4.2.2 Petri Net Model

Petri nets evolved as mechanisms for interprocess communication. They can model interaction between different components. It is this which makes Petri nets useful in protocol modelling. As seen in the previous section, FSMs were used to model each entity and the Channel individually. The state space of the global FSM was derived as a cartesian cross product of the individual set of states but the protocol state space was a subset of the global one. This difficulty arose as FSMs do not have a mechanism to explicitly model interaction between processes (an extension, communicating FSM, CFSM has been proposed).

The Petri net model of protocols is derived in two steps [Dia82]:

1. Each entity and the channel is modelled as a state machine or a net.

2. The components are connected together by explicit interconnection mechanisms to give the global system model. The interconnection mechanism corresponds to the actual implementation.

In the approach given in [Boc78], each entity and the medium is modelled as a FSM and then a Petri net model is built by connecting the machines together. This was illustrated by an example reproduced here in Figure 4.16. Here the interconnection is by sharing transitions. The number of messages in the medium can be N. This can be achieved by placing N tokens in the place denoting empty channel.

For simple models of communication medium a number of interconnection mech-
Figure 4.16: The global model using a simple interconnection mechanism
anisms have been proposed [Dia82]. The basic ones are given below:

Shared Place A place shared between the sender and the receiver processes represents the medium. It represents the fact that the message has been sent by the sender and not yet reached by the receiver. This is really the actual behavior denoting the fact that the message is in transit in the medium. Introduced by [Mer74,Dan77] it has been the most used interconnection mechanism [Dia82,BT82]. This mechanism is illustrated in Figure 4.17. A shared place representation implies potentially unbounded buffering, the sender is released after sending the message.

Merged Transition This technique was suggested in [LC75,Hoa78]. The sender and the receiver processes share a transition to exchange a message. This implies no buffering, direct transfer and a synchronized send/receive. It is illustrated in Figure 4.18.

Request-Acknowledge This mechanism accommodates the limited buffering of the receiver. It is based on the shared place interconnection and is shown in Figure 4.19. The PAR protocol represented earlier uses a variant of this scheme.

The shared place is the basic interconnection mechanism for processes belonging to distinct entities. It corresponds to the deterministic system of sequential processes [SB88]. This will be dealt with in detail in the next chapter. More advanced models of medium have been considered. These include a FIFO model [AAB78]. These have been modelled using Predicate-Transition nets and Predicate-Action nets. Other efforts include identifying messages in transit using colored Petri
Figure 4.17: The global model obtained using shared place mechanism
nets. However, in the opinion of Diaz [Dia82], as a first step, protocols should be validated with the shared place model for every exchanged message as the corresponding graph model imposes very few constraints on the medium. This approach has been widely used [Dan80,AAB78,Mer74]. As a next step, specific medium can be accounted for (FIFO etc.) using Predicate-Transition nets [BT82,Vos80,GLT79,GL79].

It will be seen in the next chapter (on implementation) how this information about modelling helps in implementation. If the abstract model of interconnection is not known then it has to be derived from the design. Finally, the Petri net model of the protocol, PAR presented in the previous section is given in Figure 4.20. The model presented is a slight modification of the one given in [Tan87]. As in the FSM model, the sender has two states, \textit{wait}_{ck} and \textit{wait}_{ck1}. The receiver also has two states, \textit{expect}_{0} and \textit{expect}_{1}. The channel has four states denoting what it carries, \textit{frame}_{0}, \textit{frame}_{1}, \text{Ack} and \text{empty}. The channel acts as shared place between the sender and the receiver processes. The scheme used in the protocol is similar to the request-acknowledge scheme as for every message sent the sender waits for the acknowledgment. A reachability analysis of the Petri net yields ten states. These are shown in Figure 4.21. The number of states is the same as the one obtained by the FSM approach. However, in contrast to the global FSM here no extraneous states are present here. The global model is obtained with relative ease too.
4.2.3 Protocol Controllers

In the previous two sections Petri nets and their application in protocol modelling was discussed. The current approach is to validate the correctness of the protocol from the Petri net model and once the design is found to be correct it is implemented in software. Work has been done to translate the Petri net directly into software to yield an implementation [Par90,III89,Raz87]. In fact one of the requirements of a good specification is that a direct implementation should be possible.

Advances in fiber optics have resulted in increased communication speeds. The links can now operate at speeds of several Gigabits per second, faster than the internal busses of fast mainframe computers. The trend towards ISDN and Multimedia systems require high speed traffic handing. All this severely limits the communication processing by software on general purpose computers [KS89]. With these things in mind handling of communication processing by a dedicated hardware is desirable. This can be done by relatively inexpensive (as compared to the processing power and time of the host computer) VLSI circuit.

A Protocol Processor is typically a coprocessor VLSI chip or chip set that relieves the host computer from communication processing functions. A Protocol Controller is a VLSI circuit that implements the functions of one or more protocol layers. A processor is programmable if it can implement different protocols or variations of the same protocol.

Physical layer protocols e.g. RS-232 have always been implemented in hardware
(they must be!). Several implementations of the Data Link layer protocols exist, [Eri85] implements the complete layer. In [AUT*86] implementation of network layer is found. Attempts have also been made to implement Transport layer protocols [Che87]. No hardware implementations exist for Session and higher layers [KS89]. However, most of the work is restricted to Data Link, Network and Transport layer protocols. This work too considers these as the target but the technique used could be extended to other levels too. Having seen that Petri nets have been used to model protocols and the fact that hardware implementations of protocols exist, it is natural to ask whether the protocols can be directly implemented from their Petri net model and whether this process could be automated. This work attempts to answer this question.

In order to give a clear idea of what is meant by a Protocol Processor and a Controller and where the present work fits, a brief sketch of two of the implementations, a Programmable Protocol VLSI Engine (PROVE) [KS89] and Protocol Engine [Che87] are presented.

PROVE It is a programmable protocol processor. The protocol is specified as a collection of communicating FSMs (CFSMs). It can implement connection oriented protocols. The architecture of the processor consists of three major blocks – Message Parser (MP), Message assembler (MA) and the Central Control Unit (CCU) Figure 4.22. The message parser breaks the header information in each incoming packet into it’s components and gives them to the CCU. The syntax of a valid packet is specified as a tree grammar and stored as a microprogram for MP and MA. The CCU implements the core
of the protocol specified as CFSMs in a formal language specification. These
are stored as tables in the external RAM. The CCU uses tokens from the MP
as inputs and performs semantic processing on them. Buffer management
and DMA interface are provided by separate units.

Within this framework, the proposed work can replace external RAM storing
the CFSMs. As an automated synthesis procedure is suggested for the im-
plementation of Petri nets the hardware can be easily changed in case there
is any change in protocol or another protocol is to be used.

Protocol Engine This implements the functions of the Network as well as the
Transport layer provided by the XTP protocol [Che]. It is aimed at Intern-
et applications. The architecture is shown in Figure 4.23. It consists of
control logic, the address logic, buffer logic, network interface logic and host
interface logic. The controllers for the various logic units are present in the
control logic unit. It also contains a FSM for the XTP protocol. This imple-
mentation uses microcode implementation and hence can be reprogrammed.
The address logic performs address translation for routing, buffer logic han-
dles both memory access needs of host and the protocol engine and transfers
data between internal shift registers and external buffers. Design of network
interface logic depends on the choice of network and of the host interface
logic on the host processor. Some interesting features include:

- 100 Mbits/s bandwidth which can be used in fiber optic links.
- adaptability to different physical layers including Ethernet, IEEE 802.3
  LAN, FDDI ring etc.
useful for implementing real time gateways between networks.

The proposed work would be used in the control logic of such a system. As in the implementation (discussed in the next chapter) Petri nets are converted into FSMs the proposed method can be modified to get a microcode implementation, if desired. However, with standardization and availability of verification tools at the design stage has lowered the probability of error in the design of protocols. Therefore programmable protocol controllers might not be very useful especially as we wish to obtain higher speeds.

This chapter provided an overview of how Petri nets, protocols and protocol controllers are related. The next chapter deals with the implementation of the Petri net model of communication protocols.
Figure 4.18: The global model obtained using merged transition mechanism
Figure 4.19: The global model obtained using request-acknowledge mechanism
Figure 4.20: The Petri net model of the protocol PAR.
State is a marking shown below as the places having tokens

0. wait_ack_0,m_0,expect_0
1. ack,wait_ack_0,expect_1
2. wait_ack_0,timeout,expect_0
3. wait_ack_1,m_1,expect_1
4. wait_ack_0,timeout,expect_1
5. wait_ack_1,ack,expect_0
6. wait_ack_1,timeout,expect_1
7. wait_ack_0,m_0,expect_1
8. wait_ack_1,timeout,expect_0
9. wait_ack_1,m_1,expect_0

Figure 4.21: The reachability graph of Petri net model of the protocol PAR
Figure 4.22: The architecture of PROVE
Figure 4.23: The architecture of Protocol Engine
Chapter 5

Implementation of
Communication Protocols: A
Case Study of Distributed
Systems

The previous chapter discussed the use of Petri nets in modelling of communication protocols and protocol controllers. This chapter presents a method of implementation of protocol controllers from their Petri net model. The implementation is in the form of a PLA which is suitable for VLSI. This method can be automated to realize a Petri net based DA system. Examples drawn from literature illustrating the proposed method are discussed.

The implementation method can be logically divided into two phases —
1. **Frontend**, it takes the Petri net model of the protocol and produces a FSM for each of the controller entities and the

2. **Backend** which generates a PLA implementation of the FSM.

## 5.1 Frontend

This phase consists of two basic steps,

1. decomposition of the protocol model into two entities representing the $N^{th}$ layer

2. for each entity, obtaining the FSM for it’s implementation

### 5.1.1 Decomposition of Protocol

In the previous chapter on modelling, the approaches to modelling were considered. The basic approach was to model each of the entities and the channel by an FSM. The Petri net model was built by representing each FSM by it’s Petri net equivalent and the overall model was obtained by connecting these together explicitly using a shared places or merged transitions [Boc78]. For a simple model of the channel, the interconnection mechanism by itself represented the channel [Dia82].

In many applications, apart from communication protocols, concurrent/distributed systems are made of synchronized sequential processes. This area of designing concurrent/distributed systems by composition of sequential components has been
the focus of active research [YFW89,DG84,JV79]. Hoare [Hoa78] suggested that channels can be used by the sequential processes to communicate and thus in building of cooperating sequential processes. However, the fact that the individual sequential processes are live (for the subclass of Petri nets, State Machine, for liveness strongly connected is the necessary and sufficient condition [Mur89]) does not guarantee that the overall system obtained by interconnecting processes using shared places will be live. The overall system may have a deadlock due to interlocking of processes. This is illustrated in the Figure 5.1 from [Ber87]. Each of the nets, $N_1$ and $N_2$ are live and safe, however, the composite net obtained by merging transitions $t_1$, $t_4$ and $t_7$ is deadlocked as shown in Figure 5.2. Both the shared place and the merged transition mechanisms can be shown to be equivalent in terms of liveness and boundedness properties. The reduction rules by Kwang et al. [LFB87,LF85] do not alter the number and direction of flow of tokens and hence do not affect liveness and boundedness of the net. Using the reduction rule RSN-4T [LF85, page 276] a shared place can be converted into a merged transition representation where the merged transition represents a macro-transition. This is shown in Figure 5.3. Using the reverse expansion a merged transition can be transformed into a shared place without affecting the liveness and boundedness. Although both representations are equivalent, in a sense, using both simultaneously is not recommended as it is difficult to handle them semantically and theoretically [Ber87]. This is understandable from the point of view of the interconnection service they provide. A shared place represents unbounded buffering whereas a merged transition represents synchronized transfer and direct transfer without buffering (refer to Chapter 4). In terms of protocols, it has been
Figure 5.1: Deadlock: The interlocking of processes
Figure 5.2: The partial reachability tree of composite Petri net of Figure 5.1 showing deadlock
Figure 5.3: The reduction RSN-4T reduces a shared place to a merged transition
suggested that shared place be used for modelling the protocol of a layer (the interaction of distributed entity) and that merged transition be used to model interface protocols (or, equivalently, interfacing of protocol levels) [Dia82].

Having seen that the composition of live State Machines using shared place (or merged transition) does not necessarily produce a live net and some stronger restrictions are needed. The protocol demands liveness as a requirement for correctness [Sid82]. Moreover, the decomposition of the overall system, in general, into interacting components is difficult. Equivalence relations are related to the composition/decomposition. Depending on the composition technique, two types of decompositions exist:

S-decomposition It was defined for merged transition composition and constitutes a partition on the places. The first work on state machine decomposition was presented by Hack [Hac72] from his work on live and safe Free Choice nets (FCN). He presented the theorem that a live and safe free choice net (LSFC) can be decomposed into interacting state machines as well as marked graphs. This property was shown to break down for higher classes of Petri nets [TV84b,TV84a]. These can be calculated using invariant analysis (S and T invariants), however, the calculation of minimal invariants poses some problems [KJ87,MS82]. Later Hack gave the definition of state machine allocable nets [Hac74] and B-equivalences were defined by André [And82,And83].

T-decomposition It is defined for shared places and constitutes a partition on transitions. De Cindio et al. defined equivalence for 1-safe superposed au-
tomata nets (a subclass of state machine decomposable nets of Hack [Hac74], with one token per component and all transitions have same number of inputs and outputs) [CMS85].

Although some theoretical results are available on state machine decomposition yet their application is difficult and no algorithms exist. This has led to search for subclasses which provide easy decomposition. One such subclass of cooperating sequential processes, deterministic system of sequential processes (DS), was introduced and analyzed by Reisig [RMB83]. This class was shown to be adequate to model communication protocols. It also exhibits some desired properties with regard to liveness of the composite model [SB88]. For this study three different cases for decomposition are considered:

- the composite model of the system is known with the entities demarcated
- the composite model is a DS and the entities are not demarcated
- the composite model is not a DS and the entities too are not known

Case 0

The implementation of the protocol from the Petri net model can be integrated into the design-analysis cycle. The Petri net model is built using the structured approach of modelling each entity by an FSM and the channel is represented by shared places or merged transitions. The overall system can then be analyzed and when the design is found to be correct the task of implementation can be initiated. With this scenario, it is realistic to assume that the composite protocol model is known along with the information about the models of the entities, i.e., the model
with entities demarcated is known. This corresponds to the existing approach in literature where the overall model is given demarcated [SKC90,MR86]. The task of automated implementation then consists of detection of channel and building a FSM for each component. With the knowledge of entities, the task of channel detection reduces to finding the common places or transitions.

Procedure 1 (Break Entities)

Input The Petri net model of the protocol with each entity demarcated in the protocol

Output Petri net model of each isolated entity

begin

for each entity

if shared place mechanism is used then

Search for places which are common to both entities

else if merged transition is used then

Search for transitions that are common to both entities

end if

if shared place mechanism is used then

for each transition

for each arc connected to shared places

if the arc is connected from the transition
to the shared place, $x$ (output arc) then

assign an output, $x$ to the transition, ($x$)

and cut the arc connecting the transition to shared place

end if

if the arc is connected from the shared place

to the transition, $x$ (input arc) then

assign an input condition, $x$ to the transition, $< x >$

and cut the arc connecting the shared place to the transition

end if

end for

end for

if merged transition mechanism is used then

for each merged transition

for each arc connected to merged transition

if the arc connected to the merged transition

is an input arc and it comes from place $v$

which belongs to the other entity then

assign an input condition, $< v >$ to the transition

and cut the arc

else if the arc connected to the merged transition

is an output arc and it comes from place $w$

which belongs to the other entity then

assign an output, ($w$) to the transition
and cut the arc

end if

end for

end for

end if

the marking of the isolated process is the original
marking of the composite net restricted to the places
belonging to the entity

end for

end

By the application of this procedure, the Petri net is separated into components representing each entity. The Petri net obtained is annotated with input conditions and outputs associated with the transitions. This is not different from the Petri nets defined earlier. They just represent a modelling convenience. The input condition on the transition is equivalent to a source place connected to the transition and models the interface to the external world. Figure 5.4 shows the different steps of the procedure.

Definition 12 This representation, called Extended Petri net has been used earlier by Yoeli, [Yoe82] and formally defined as: A Extended Petri net (EPN) is an 8-tuple, \( E=(P,T,F,M_0,D,C,A,d_0) \) where,

\( P,T,F,M_0 \) are as defined earlier for ordinary Petri nets,

\( D\): is a set of data, from external world,
Place of Entity 1 and 2 resp
channel
place

Figure 5.4: Illustration of transformations in procedure Break_Entity
Figure 5.5: Application of procedure Break_Entities on the Petri net model of PAR

\[ d_0 \in D \text{ represents initial data} \]

\[ C: T \times D \rightarrow \{0,1\} \text{ is the (input) condition function} \]

\[ A: T \times D \rightarrow D \text{ is the action (output) function} \]

The firing rule is modified such that a transition is enabled if all of its input places have tokens and all the input conditions are true (\( C(t,d)=1 \)). The firing of a transition deposits tokens in each of the output places and asserts each of the outputs associated with it (\( A(t,d)=d' \)).

Example 1 An application of the procedure Break_Entities for the running example PAR is given in Figure 5.5.
Case 1

In this case it is assumed that the composite model is known but the entities are not demarcated. However, the composite model is built according to the restrictions imposed by Reisig [RMB83] and is a deterministic system of sequential processes, DS. Firstly, the definition of DS is presented. This is done using modified definitions presented by Souissi [SB88] using the terminology defined earlier and is slightly different from the one given by Souissi. Next, the decomposition rules are presented and finally, a brief mention of the desirable properties of the DS is made.

Definition 13 (Isolated Process) A Petri net is an isolated process iff it is a state machine, i.e.

\[ t = t^* = 1 \forall t \in T \]

Definition 14 (Sequential Processes) A Petri net \( G = (P \cup K, T, F) \), \( K \) is a set of channels, is a sequential process iff

1. \( P \cap K = \emptyset \)

2. the subgraph generated by \( (P \cup T) \) is a SM

3. \( \forall p \in P \ p^* > 1 \implies \forall t, t' \in P^* \ t = t' \)

To model non-isolated processes, places are added, called channels, to the state machine. They are connected via arcs to transitions of SM. Based on the marking of the SM two cases are possible, one in which there is only one token in the Petri net (S-marking) and the other where the marking may have more than one token at a time.
Definition 15 (Deterministic System of Sequential Processes, DS) A Petri net, $S = \{ R_i = \langle P_i \cup K_i \rangle, T_i, F_i \mid i = 1, 2, \ldots, s \}$ is a set of sequential processes. $S$ is a deterministic system of sequential processes iff

1. $P = \cup P_i, K = \cup K_i, T = \cup T_i, F(x, t) = \sum_{i=1}^{s} F_i(x, t)$,
   \[
   F(t, x) = \sum_{i=1}^{s} F_i(x, t)x \in P \cup T, t \in T
   \]

2. $\forall R_i, R_j \in S \ i \neq j \implies (P_i \cup T_i \cup K_i) \cap (P_j \cup T_j \cup K_j) = K_i \cap K_j$

3. $\forall k \in K \exists i, j \in I \ T_i \supset \ast k, T_j \supset \ast k$

Condition 1 defines DS as a set of sequential processes, the second provides interconnection using channels and the third puts the restriction that any channel is shared by only two processes, one of them being the source and the other being the consumer. From the discussion at the beginning of the chapter, it is clear that given a Petri net model, decomposition into interacting components is not trivial. In case 0 it was assumed that the models of the entities were demarcated in the given overall model. If it is not so and only the overall model is available then the decomposition problem is tough in the general case. Keeping in mind that each entity can be modelled by a FSM (which can be modelled by SM subclass of Petri nets) and that, in general, connecting live components together does not guarantee that the overall system will be live, a subclass of Petri nets, DS (already defined) were proposed by Reisig [RMB83]. A method of decomposing DS automatically was proposed by Souissi [SB88]. This was implemented by them by an expert system, in the language, NIMA [Bel88]. Their decomposition method is presented below:

Procedure 2 (Souissi)
Input  The Petri net model of the protocol which belongs to the subclass DS

Output  Petri net model of each isolated entity

begin

Check whether the Petri net belongs to the DS class

for every place

    find if it is a state place or a channel place

delete the channel places and find the connected components

check that every channel comes from the same connected

    component and goes to another connected component end

The detection of Petri net being a DS is done by the a set of rules, for instance:

Rule 1 (Souissi)  If two places, \( x \) and \( y \), are input places of a transition with no
other inputs and neither \( x^* \subseteq y^* \) nor \( y^* \subseteq x^* \) then the net is not a DS

Similarly, the task of finding state places and channels is done by a set of rules:

Rule 1 (Souissi) Any place that is a single input or a single output place of a
transition is a state place.

Rule 2 (Souissi) If a transition with two inputs, \( p \) and \( q \), and no other entry has
\( p^* \subseteq q^* \) then, \( p \) is a state place and \( q \) is a channel.

Rule 3 (Souissi) If a transition has an input (output) place already identified as
a state place then the other input (output) places are channels

Addition of another rule to this set could make the task of finding channels easier:
Figure 5.6: Application of procedure of Souissi on the modified Petri net model of PAR without loss being explicitly modelled

Rule 1 If a place is identified as a channel then all the arcs connected to it are deleted.

Once the channel places have been identified, the method presented in Case 0 can be applied to obtain annotated Petri nets representing each entity.

Example 2 An application of the procedure of Souissi on the modified Petri net model for the running example PAR is given in Figure 5.6. Here the loss is not explicitly modelled. The transition labelled 'timeout' models loss.

Some of the attractive properties of this class are presented in [SB88]. These include results on liveness of a DS with respect to its components, monotony of liveness for structured DS and the fact for structured DS, for any initial state of
it's processes there exists an initial configuration of the channel such that the net is live. These make the task of verification of properties of protocol easier. This class is amenable to decomposition and hence is suitable from implementation aspects.

Case 2

The protocol is modelled as FSMs interconnected to obtain a global model using Petri nets. Only the global model is available. There is no information about entities, the net is not DS and neither can be transformed into one [SB88].

This scenario will be treated as a special case of the more general problem of finding interacting components of concurrent/distributed systems which is handled in the next chapter.

The treatment of the three cases concludes the first step of the frontend decomposition. The next section treats generation of FSM from the annotated Petri net model.

5.1.2 FSM from the annotated Petri net

The process of obtaining the FSM from the annotated Petri net has two steps:

1. refinement of the entity

2. generating an FSM for the underlying unannotated Petri net.
Refinement

As discussed in the previous chapter on modelling, each modelled entity contains some local actions which occur within the entity itself. These do not involve any communication with the peer but are related to the correct operation of the protocol (e.g., timer in a data link protocol). These can be modelled as always true inputs with respect to the protocol. However, at the time of implementation of an entity, these have to be interpreted as local conditions. Refinement here means that the transitions which do not have any input conditions or output conditions and are used to model local conditions are associated with locally (to the entity) inputs and outputs. This practically means that signals are coming from other constituents of the protocol processor (refer to Section on protocol controllers in Chapter 4).

Moreover, there is one more type of signals coming to the controller, from the upper layer. As the interface and service specifications are usually treated separately from the protocol specification this information is usually missing. In process of refinement this information has to be added so that the controller for the protocol layer can be built.

One way of handling refinement for local actions is to associate labels to transitions signifying local actions. In the protocol analysis stage these labels are ignored and no interpretation is associated with them, however, in the implementation stage these are interpreted as input conditions and outputs. Alternatively, this can be done at this stage (after entities have been separated). In the Petri net
model of protocol PAR presented 4.20 (chapter 4), the place timeout was added to get a deterministic overall model relating the loss and the firing of transitions. Timer being a local action for the data link layer, Tannenbaum [Tan87] did not model it and just represented it as a label. The model without the ‘timeout’ place was used in Figure 5.6 to illustrate decomposition of DS.

The task of refinement for interface specification is more complicated. The interface specification depends on the protocol being used at the higher layer. In order to ensure compatibility and at the same time allowing vendors the freedom of choice of protocols, the OSI model does not fix the protocol for each layer. Instead what has been done is to fix the functions of layers. So for the same protocols of layer N, different protocols of layer N + 1 are possible. For compatibility reasons the interface protocol is treated separately and can be implemented as an ‘upper layer interface unit’ (refer to Protocol Engine in chapter 4). The implementation of such a unit can also be done along the same lines as protocols.

However, if the upper layer protocol is known then the interface and the protocol specification can be put together and an integrated interface- protocol model can be built during the design process. This is illustrated in Figure 5.7. Then as a first step the integrated model would be separated into the interface Petri nets at each entity and an annotated Petri net obtained for the protocol model. In the analysis phase do not play any part and are just treated as labels. When this model is input to the implementation phase these would be interpreted as conditions and outputs. The other approach could be to add to the annotated Petri net obtained
Figure 5.7: Integrated interface-protocol model
after decomposition of protocol the interface information as refinement.

**FSM from annotated Petri net**

The reachability graph of the underlying unannotated Petri net defines an FSM [Val81b]. The conditions and the output are used then to replace the transitions labelling the arcs of the transition. This then defines a FSM. Two theorems are presented to back this assertion.

**Theorem 1** For every isolated entity a FSM can be obtained.

**Proof** The proof is based on two previous results and the property of protocols.

It can be assumed that the protocol to be implemented has been verified to be correct. The protocol design rules require that the protocol satisfies the following properties [Sid82]:

- liveness
- boundedness
- completeness
- deadlock freeness
- terminating or cyclic behavior
- absence of non-executable interactions
- livelock freeness

This means that the given overall protocol model will be live and bounded.

The overall system can then be partitioned into subnets using boundary
transitions (the partitioning technique of [Fah90]) of one entity. These transitions are the ones that are connected to the shared places. These satisfy the restriction imposed that the boundary transitions should have arcs going to both subnets. This is illustrated in Figure 5.8. With this partition each subnet is in fact an entity. According to [Fah90] when the partition used satisfies the constraint (of boundary transition) then the properties of the subnets would be the same as those of the net. Thus, the analysis of subnets would yield the same results on behavior as the original net. This, he suggested could be used to analyze big nets by partitioning and examining the properties of subnets. In this case, the properties of the net are known to be live and bounded. Hence, the subnets too are going to be live and bounded.
Having shown that the subnets representing entities are live and bounded, a result from Valk [Val81b] can be used to show that for every regular net, bounded being a proper subclass of regular nets, the reachability graph defines a finite automaton.

Theorem 2 (VALK) A Petri net $PN=(P,T,F,M_0)$ is regular iff

$$\exists k \in \mathcal{N} \ \forall M \in R(N,M_0) \ \forall M' \in R(N,M_0) \ \forall p \in P \ \forall p' \in P \ \ M'(p) \geq M(p) - k$$

The proof of this theorem was given by constructing a finite automaton $\mathcal{F} = (Q, \Sigma, \delta, q_0, F)$, where,

- $Q = \{ M \in \mathcal{N} \mid M(p) \leq c \} \cup \{ q_g \}$ is the set of states,
- $\Sigma = T$, the input alphabet, is the set of transitions,
- $\delta : Q \times \Sigma \rightarrow Q$, is the state transition function,
- $q_0 = M_0$, is the initial state,
- $F = Q - \{ q_g \}$, is the set of final states with $q_g$ the garbage state,

This finite automaton $\mathcal{F}$ accepted the language produced by firing sequences, $F(N)$ defined as,

$$F(N) = \{ w \in T^n \mid M_0(w) > \}$$

Theorem 3 (VALK) There is an effective procedure that associates to every regular Petri net $N$ a finite automaton accepting $F(N)$.

The procedure is the construction of the automaton of the previous theorem. Using these results from Valk [Val81b] and the fact that each entity subnet is
bounded, a FSA can be obtained. In the obtained FSA the arcs are labelled by transitions. These are then replaced by input conditions and outputs associated with the transitions.

The procedure of obtaining an FSM from the annotated Petri net is summarized below:

Procedure 1

begin

for each entity

    find the reachability graph of underlying unannotated Petri net,
    \( RG=(V,E) \), where,
    Nodes of the graph, \( V=\mathbb{R}(M_0) \)
    The set of labelled, directed arcs, \( E \) has an arc from \( V' \) to \( V'' \)
    labelled by \( t \) if
    for the corresponding markings, \( M' \) and \( M'' \), \( M''|t > M'' \)

    do for the reachability graph

    for every arc

        substitute the transition labelling the transition, \( t \)
        by the input conditions, \( x, y, \ldots \) and
        outputs, \( a, b, \ldots \) associated with the
        transition as, \( < x, y, \ldots > /(a, b, \ldots) \), where,
        all the conditions \( x, y, \ldots \) should be satisfied
        for the transition to occur and the change produces
        outputs \( a, b, \ldots \). This is finite and isomorphic to a FSM.
Figure 5.9: The FSMs obtained for the sender processes of the protocol PAR

Example 3 This step of the implementation is illustrated by the running example PAR shown in Figure 5.9.

5.1.3 Implementation Aspects

This section discusses the different issues involved in the implementation of Petri nets. The basic tool for implementation was P-NUT (Petri Net UTilities) developed at the University of California at Irvine. The subsequent sections discuss the different aspects:

Specification

P-NUT accepts a Petri net in textual form (a graphic editor exists but is not currently installed as it requires dedicated workstation) with one record for every
transition. The general format of the record is –

\[[\text{Data\_dependent\_Conditions}]\cdot\text{name\_of\_transition}\cdot\text{input\_places} \rightarrow \text{output\_places}\{\text{actions}\}\]

The data dependent conditions and actions are optional. The conditions can be specified using boolean operations whereas for actions arithmetic assignments as well as boolean expression can be given. The input\_places is a list of input places to the transition under consideration separated by commas. Similarly the output\_places is a list of output places of the transition separated by commas. Source/sink transitions are allowed. Multiple arcs to a transition are specified by giving the number within braces after the place name. This weight is 0 for inhibitor arcs. The initial marking of the net is specified as the last record by:

\[<\text{place}(\text{number}), \text{place} \ldots>\]

The place having more than one token is shown by giving the token count after the place name in braces. The subset of P-NUT tool used in this work is shown in Figure 5.10. The textual form of Petri net is input to the translation program, \textit{transl}, which converts it into an intermediate representation (ASCII file). In this format there is a record for each place/transition with the following fields:

\textbf{Place record := Place: an internally generated integer}

\begin{enumerate}
\item \textbf{Name: name of the place as given in the specification}
\item \textbf{Tokens: the initial token count}
\end{enumerate}

\textbf{Transition record := Transition: internally generated integer}

\begin{enumerate}
\item \textbf{Name: as given in the specification}
\item \textbf{Input\_places: the internal integers representation of the input places separated by commas}
\end{enumerate}
Figure 5.10: The subset of P-NUT tool

Output_places: the internal integers representation
of the output places separated by commas

Predicate: the boolean expression given
as condition on the transition
with a ‘&’ represented as ‘and’ and so on

Action: for boolean expressions it
is treated as for Predicates whereas
for arithmetic assignments it outputs
the given expression.

Frontend Implementation

The method for decomposing for case 0 is given below:

Procedure 3 (Decomposition (Case 0))
Input  Textual representation of protocol with the entities demarcated.

Output  Annotated Petri net model of each entity separated.

1. For entity 1 obtain the translated internal representation of the Petri net.

2. Build a table for places, transitions used in entity 1.

3. For entity 2 obtain the translated internal representation of the Petri net.

4. From the internal representation of net (for entity 2)

   for each place
   
   look in the table for entity 1
   
   if it occurs in entity 1 then
   
   put it in the table for shared places
   
   end if
   
   end for

   for each transition
   
   look in the table for entity 1
   
   if it occurs in entity 1 then
   
   put it in the table for shared transitions
   
   end if
   
   end for
5. Assuming that only one of the interconnection mechanism is used, only one of the tables, shared place or shared transition will non-empty.

if shared.place is non-empty then
    set interconnect = shared
else if merged.transition is non-empty then
    set interconnect = merged
end if

do delete the empty table

6. Go back to the original Petri net textual representation

for each entity
    if shared=1 then
        for each transition do
            for each shared place do
                if the common place occurs to the left of transition only then
                    move it to the predicate list,
                    ANDing it with the terms present
                else if the common place occurs to the right of transition only then

...
move it to the action list, ANDing it with the terms present

end if

end for

else if merged=1 then

for each merged transition do

for each place connected to it do

if it is an input place and belongs to the other entity then

move the place to predicate list

ANDing to those already there

else if it is an output place and belongs to the other entity then

move the place to action list

ANDing to those already there

end if

end for

end for

end if

end for

The method for obtaining an FSM for each entity is given next. It is assumed that refinement is handled separately for interface specification and for the local action
it is assumed that the original model contained the required inputs and output as predicate/action for transition but were ignored in previous processing.

Procedure 2 (FSM Generation)

Input  Annotated Petri net model of an entity.

Output  FSM for the entity.

1. The annotated Petri net is again translated by transl to get the modified internal representation with predicates and actions. If the local conditions were not initially added then this is done at this point.

2. This annotated Petri net is fed to the tool, rgb for generating reachability of a Petri net. This tool does not consider predicates and actions in building the reachability graph. Thus it obtains the reachability graph of the underlying unannotated Petri net.

3. The reachability graph is represented with one record per state in the following format:

   S state_number marking
   A next.state_number(transition_number)

   Where, state number is an internally assigned integer, marking is expressed as place numbers, separated by commas which have tokens in the given marking. The transition number gives the transition whose firing leads to the transition.
4. The predicates and actions associated with the transitions are available in the internal representation of the net which is included along with the reachability graph. This information will be used to interface the output of frontend to the backend automatically.

Example 4 The two procedures are illustrated for the running example of protocol PAR in Figures 5.11 and 5.12.

5.2 Backend

The backend is basically a silicon compiler for FSM. It is based on a subset of software tools made at the University of California, Berkeley, called 1986 VLSI Tool Kit [SMHO85] and VLSI Design Tools [SMHO87]. The entire process is automated. It takes a description of the state machine to be implemented and gives the layout of the PLA implementing it. What remains after that is to place the PLA in a pad frame, connect the input of FSM and and the next state outputs, route the inputs and outputs to I/O ports and the chip would be ready to be sent for fabrication. These tasks can also be done using available tools. The sequence in which various tools were interfaced are shown in Figure 5.13. Finally, the FSM can be extracted from the PLA and simulated to verify correctness of operation before being sent for actual fabrication. The sequence of tools used for this purpose is shown in Figure 5.14. A brief review of the tools is presented. Details for the tools are available in the accompanying manuals with the tools [SMHO85,SMHO87].
Figure 5.11: (a) The textual Petri net model of PAR protocol with entities demarcated (b) The annotated Petri net model of the sender process of PAR obtained by the decomposition procedure.
Petri Net: "stdin"
Places: 2
Name:
Tokens: 0
Transitions: 4
Name:
Input Places:
Output Places:
Array:
Tarray:

Place: 0
Name: wait_ack_1

Place: 1
Name: wait_ack_0
Tokens: 1

Transition: 0
Name: emit_0
Input Places: 0
Predicate: ack
Output Places: 1
Action: m_0

Transition: 1
Name: retrans_0
Input Places: 1
Predicate: timeout
Output Places: 1
Action: m_0

Transition: 2
Name: emit_1
Input Places: 1
Predicate: ack
Output Places: 0
Action: m_1

Transition: 3
Name: retrans_1
Input Places: 0
Predicate: timeout
Output Places: 0
Action: m_1

End

Reachability Graph: "stdin"
States: 2
Arcs: 4
S 0 1
A 0(1)
A 1(2)
S 1 0
A 0(0)
A 1(3)
End

Figure 5.12: The reachability graph for the sender process of PAR obtained from the procedure for FSM generation for PAR.
Figure 5.13: The sequence of tools used for FSM Silicon Compiler
Figure 5.14: The sequence of tools used for simulation
5.2.1 Mealey Equation Generator (meg)

It takes a high level description of the FSM and generates equations for the outputs and the next_state variables in terms of the inputs and the and the state inputs. The input is given in the following format:

- - in a line treats the rest of text as comment

INPUTS:  input signal list separated by blanks ;
OUTPUTS: output signal list separated by blanks ;
RESET ON input_signal TO [STATE] reset_state ;
- gives the signal which forces the machine to
  reset_state and produces the specified output

Here, ready will be used as the reset signal and the output produced signifying that the machine has been forced to reset_state is begin. After this preamble, the machine is described with one record for each state. The format of the record is given by:

state_name : control ;

The state_name is the symbolic name of state made of alphanumeric characters with the first character as a alphabet. The control portion is specified in one of the three ways listed below:

IF [NOT] input THEN next_state ELSE next_state ;

GOTO next_state ;

CASE (input selection list)
cases

ENDCASE [default];

The next state is of the form:

state_name [ (output_signal_name) ]

The output signal is specified as:

output_signal [=0|1|?]]

where, 0 makes the signal low, 1 high and ? means a don’t care. If no value is specified, the signal is made high.

The case statement specifies a bit pattern for the variables listed as conditions for next state assignment. The case selectors are applied one at a time from top to bottom till a match is made with a bit pattern and the control is then transferred to the specified state, producing the required output. If no match occurs, the control is transferred to the default state. If no default state is given an error occurs.

Apart from this, meg can accept symbolic names for inputs and outputs defined through the #define statement. These are automatically expanded by invoking the C language preprocessor.

The task of automating the process of implementation directly from Petri net model requires the output of the frontend to be interfaced with the input of the backend. This can be achieved by the following procedure:
Procedure 4 (Fr_Ba_Interface)

Input  The annotated Petri net model of an entity.

Output  meg input description.

1. From the intermediate representation of the Petri net make a list of all the predicates on transition and put them as INPUTS and add ready.

2. From the intermediate specification make a list of all actions and put them as OUTPUTS of the meg input specification and add begin.

3. If the same signal, s, appears in both input and output statements then create a new signal, s*, and replace s by s* in the output list. Also add s and s* pair in a table containing the present state input and next state output pairs. These pairs will be connected in the generated PLA.

4. Make the RESET statement with the reset signal as ready and the associated output, begin. The reset state is defined as S.0.

5. for each transition, symbolic names for the predicate associated with it, t.no.i, and the action associated with it, t.no.o, are created. The number associated is the internally assigned number for the transition. Another variable, t.no.ni is created. It has an integer value equal to the number of inputs in the predicate product term associated with the transition.

6. the maximum size of the product term associated with a transition as predicate is found.

7. the following code is added to the meg input specification:
begin for every transition

  do create symbolic definitions
      
      #define t.no.i predicate_list
      
      #define t.no.o action_list
      
      #define t.no.ni 1 1 ... 1
      
  end do end for

8. The predicate(action) list is the conditions(outputs) associated with transition separated by blanks. The definition for t.no.ni is an integer coding of the integer variable i.e. the number of 1's equal the integer value.

9. an entry is created in the meg input specification as using the reachability graph:

begin for each state do

  create a record S.no : CASE

  where no is the internally generated state number

  for each arc going out of the state do

    add the symbolic name of transition which labels it,
    t.no.i for t.no to case statement list

    add a case selector, of the form
    t.no.ni = > S.next (t.no.o)

  end for end for
sender process
INPUTS: ready ack timeout;
OUTPUTS: begin emit_0 emit_1 err;
RESET ON ready TO SP1(begin);
SP1: CASE(ack,timout)
    1 0 => SP2 (emit_1);
    0 1 => SP1 (emit_0);
    0 0 => LOOP;
    ENDCASE => error(err);

SP2: CASE(ack,timout)
    1 0 => SP1(emit_0);
    0 1 => SP2(emit_1);
    0 0 => LOOP;
    ENDCASE => error(err);

error: GOTO LOOP(err);

Figure 5.15: The input meg specification for sender process of PAR protocol

This procedure provides an interface between the frontend and the backend to allow automated synthesis. If the refinement for local conditions is done at the very outset, the frontend and the backend provide complete automation of the implementation. The possibility of minimizing the state machine generated using standard methods is open and tools can be interfaced to minimize the state machine before it is input to the backend.

Example 5 The input meg specification for the protocol PAR as presented in Figure 5.15 and the output produced is shown in Figures 5.16 and 5.17.

5.2.2 eqntott

The meg program outputs the equations for the output and next_state variables in terms of inputs and previous_state variables in a two level AND- OR format suitable for implementation by a PLA. The option for FSM is used. These can
SUMMARY INFORMATION GENERATED BY MEG FROM FILE esender

meg: Number of inputs = 3
meg: Number of outputs = 4
meg: Number of states = 3
meg: State field size = 2 * 2
meg: Table width = 11

INPUTS:
  i00: ready
  i01: ack
  i02: timeout
  s00: StBit0* (msb)
  s01: StBit1* (lsb)

OUTPUTS:
  n01: StBit1* (lsb)
  n00: StBit0* (msb)
  o00: begin
  o01: emit_0
  o02: emit_1
  o03: err

State Table

<table>
<thead>
<tr>
<th>i</th>
<th>i</th>
<th>i</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>n</th>
<th>n</th>
<th>n</th>
<th>o</th>
<th>o</th>
<th>o</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1 - - - 0 0 1 0 0 0 ANY

| 0 0 0 0 0 0 0 0 0 0 0 0 | SP1 |
| 0 1 0 0 0 1 0 0 0 1 0 0 | SP1 |
| 0 0 1 0 0 0 0 0 1 0 0 0 | SP1 |
| 0 1 1 0 0 0 1 0 0 0 0 1 | SP1 |
| 0 0 0 0 1 1 0 0 0 0 0 0 | SP2 |
| 0 1 0 0 1 0 0 0 1 0 0 0 | SP2 |
| 0 0 1 0 1 1 0 0 0 1 0 0 | SP2 |
| 0 1 1 0 1 0 1 0 0 0 0 1 | SP2 |
| 0 - - 1 0 0 1 0 0 0 0 1 | error |

meg: Eliminated Vectors = 151824

Figure 5.16: The output of meg for sender process of PAR protocol in form of truth table
INORDER=
  ready
  ack
  timeout
  StBit0*
  StBit1*;
OUTORDER=
  StBit1*
  StBit0*
begin
emit 0
emit 1
err;
StBit1*=
  (!ready&!ack&timeout&!StBit0*&&StBit1*)|
  (!ready&ack&!timeout&!StBit0*&&StBit1*)|
  (!ready&ack&!timeout&!StBit0*&&StBit1*);
StBit0*=
  (!ready&StBit0*&&!StBit1*)|
  (!ready&ack&timeout&!StBit0*&&StBit1*)|
  (!ready&ack&timeout&!StBit0*&&!StBit1*);
begin=
  ( ready);
emit 0=
  (!ready&ack&!timeout&!StBit0*&&StBit1*)|
  (!ready&!ack&timeout&!StBit0*&&!StBit1*);
emit 1=
  (!ready&!ack&timeout&!StBit0*&&StBit1*)|
  (!ready&ack&!timeout&!StBit0*&&!StBit1*);
err=
  (!ready&StBit0*&&!StBit1*)|
  (!ready&!ack&timeout&!StBit0*&&StBit1*)|
  (!ready&ack&timeout&!StBit0*&&!StBit1*);

Figure 5.17: The output of meg for sender process of PAR protocol in form of equations
be fed into eqntott which produces a truth table. This is in the form a PLA personality with one line for each minterm for input and present.state and another for the next.state and output. The elements of the table at input side are 1, 0, or -. A 1 if the variable appears uncomplimented in the minterm, a 0 if it appears complemented and a - if it does not. At the output the terms are 1,0 and x, where x stands for don’t care. This format is suitable for minimization by espresso to which it is fed next.

Example 6 The input to the eqntott was shown in Figure ???. The output which is compatible with the input of espresso is shown in Figure 5.19.
Figure 5.19: The output of espresso for sender process of PAR protocol

5.2.3 espresso

It accepts a two level description of a boolean function. It minimizes the minterms required to cover the function and hence the size of the PLA. It can also minimize multiple-valued boolean functions. This was used to minimize the PLA personality produced by the eqntott program. The results on reduction by espresso are available in literature.

Example 7 The output of the espresso is given in Figure 5.19. The number of product terms was minimized from ten to eight.
5.2.4 mpla

It is a technology independent PLA generator. The reduced PLA personality is input to this along with the desired style. The tool currently supports nmos and scmos styles. The details are available in the manual [SMHOS85].

Example 8 The PLA for the protocol PAR was generated using in nmos using buried contact, cis (inputs and outputs of the PLA on same side) style. Clocked inputs and outputs were used. Berkeley design rules were used. The PLA produced has the present state inputs and next state outputs unconnected. This can be done automatically using other available tools, e.g. cfl or an interactive editor, magic. The controller can then be put in a padframe, inputs and outputs connected to I/O pads (again, this can be done using cfl). The chip would then be ready to be sent to the foundry. This could then be used in a chip set implementing a protocol processor for a layer (or more). The layout produced is shown in 5.20.

5.3 Verification

In any design automation system, verification is important to check the correctness of design. Verification is done at different levels. In this system simulation can be done at Petri net model level [Raz87], at the FSM level, slang (currently not available here) and finally simulation can be done at the layout level. The FSM is extracted from the layout using an extraction tool mextra. The sequence of operations and the tools used are shown in Figure 5.14. The simulation is done using rnl and the output waveforms are viewed using a simulated oscilloscope, simscope.
Figure 5.20: The layout of the controller of sender process of PAR produced by mplab
Example 9 The format of the extracted circuit is shown in Figure 5.21. The timing diagram obtained for sender process of PAR is shown in Figure 5.22.

5.4 Example

In conclusion, another example drawn from literature [MR86] and the major steps are presented in Figures 5.23, 5.23, 5.25 and 5.26.

The sender process of the protocol has 11 inputs, 10 outputs and 33 states. The reduction achieved by the espresso was from 138 product terms to 98 product terms.
Figure 5.21: A portion of the output of mextra showing the extracted FSM for sender process of PAR protocol.
Figure 5.22: The timing diagram for sender process of PAR protocol
/* Sender Model */

/* sender_sends_message */
Sready, send_flag0, slot_empty0, last0 -> msg0_0, last1, slot_filled0, send_flag0, waitack
Sready, send_flag1, slot_empty0, last0 -> msg1_0, last1, slot_filled0, send_flag1, waitack
Sready, send_flag0, slot_empty1, last1 -> msg0_1, last0, slot_filled1, send_flag0, waitack
Sready, send_flag1, slot_empty1, last1 -> msg1_1, last0, slot_filled1, send_flag1, waitack

/* sender dequeues an acknowledgement packet */
waitack, first_ack0, filled_ack0, ack msg0_1 -> first_ack0, ack slot empty1, ack1
waitack, first_ack0, filled_ack0, ack msg0_0 -> first_ack1, ack slot empty0, ack0
waitack, first_ack1, filled_ack1, ack msg1_0 -> first_ack0, ack slot empty1, ack0
waitack, first_ack0, filled_ack0, ack msg1_0 -> first_ack1, ack slot empty0, ack1

/* Good ack- alternate bit and send next message */
ack0, send_flag0 -> send_flag1, Sready
ack1, send_flag1 -> send_flag0, Sready

/* Bad ack - ignore it and let timeout take care of retransmission */
ack0, send_flag0 -> send_flag0, waitack
ack1, send_flag1 -> send_flag1, waitack

/* timeout (if no acknowledgement available) and retransmit last message */
waitack, first_ack0, ack slot empty0 -> Sready, first_ack0, ack slot empty0
waitack, first_ack1, ack slot empty1 -> Sready, first_ack1, ack slot empty1

/* Receiver model */

/* Receiver dequeues message */
Rready, slot_filled0, first0, msg0_0 -> read0, first1, slot_empty0
Rready, slot_filled0, first0, msg1_0 -> read1, first1, slot_empty0
Rready, slot_filled1, first1, msg0_1 -> read0, first0, slot_empty1
Rready, slot_filled1, first1, msg1_1 -> read1, first0, slot_empty1

/* Receiver verifies message by matching with rcv_flag */
read0, rcv_flag0 -> rcv_flag1, acking, msg0
read0, rcv_flag1 -> rcv_flag1, acking, msg0
read1, rcv_flag1 -> rcv_flag0, acking, msg1
read1, rcv_flag0 -> rcv_flag0, acking, msg1

/* receiver sends an acknowledgement with flag flag received */
acking, msg0, ack slot empty0, last ack0 -> Rready, ack msg0_0, last ack1, filled_ack0
acking, msg1, ack slot empty0, last ack0 -> Rready, ack msg1_0, last ack1, filled_ack0
acking, msg0, ack slot empty1, last ack1 -> Rready, ack msg0_1, last ack0, filled_ack1
acking, msg1, ack slot empty1, last ack1 -> Rready, ack msg1_1, last ack0, filled_ack1

/* initial marking */
<Sready, send_flag0, Sready, rcv_flag0, last0, last_ack0, first0, first_ack0, slot empty0, ack slot empty0 slot empty1, ack slot empty1>
\[ \text{slot0\_empty}\]\text{Sready, send\_flag0, last0} \rightarrow \text{last1, send\_flag0, waitack[msg0 0\&slot} \\
\text{[slot0\_empty]}\text{Sready, send\_flag1, last0} \rightarrow \text{last1, send\_flag1, waitack[msg1 0\&slot} \\
\text{[slot1\_empty]}\text{Sready, send\_flag0, last1} \rightarrow \text{last0, send\_flag0, waitack[msg0 0\&slot} \\
\text{[slot1\_empty]}\text{Sready, send\_flag1, last1} \rightarrow \text{last0, send\_flag1, waitack[msg1 0\&slot} \\
\text{[filled\_ack0\&ack\_msg0 0]}\text{waitack, first\_ack0} \rightarrow \text{first\_ack1, ack0[ack\_slot0\_empty} \\
\text{[filled\_ack1\&ack\_msg1 0]}\text{waitack, first\_ack0} \rightarrow \text{first\_ack1, ack1[ack\_slot0\_empty} \\
\text{[filled\_ack1\&ack\_msg0 1]}\text{waitack, first\_ack0} \rightarrow \text{first\_ack1, ack0[ack\_slot1\_empty} \\
\text{[filled\_ack1\&ack\_msg1 1]}\text{waitack, first\_ack0} \rightarrow \text{first\_ack1, ack1[ack\_slot1\_empty} \\
\text{ack0, send\_flag0} \rightarrow \text{send\_flag1, Sready} \\
\text{ack1, send\_flag1} \rightarrow \text{send\_flag1, waitack} \\
\text{ack1, send\_flag0} \rightarrow \text{send\_flag0, Sready} \\
\text{ack1, send\_flag0} \rightarrow \text{send\_flag0, waitack} \\
\text{[ack\_slot0\_empty]}\text{waitack, first\_ack0} \rightarrow \text{Sready, first\_ack0[ack\_slot0\_empty} \\
\text{[ack\_slot1\_empty]}\text{waitack, first\_ack1} \rightarrow \text{Sready, first\_ack1[ack\_slot1\_empty} \\
\langle\text{Sready, send\_flag0, last0, first\_ack0}\rangle \\

Figure 5.24: The textual form of annotated Petri net model of sender of Alternating bit protocol
sender process for alternating bit protocol
INPUTS: ready slot0_empty slot1_empty filled_ack0 ack_msg0 0 filled_ack1;
OUTPUTS: begin err msg0 0 msg1 0 msg1 0 slot0_filled slot1_filled;
RESET ON ready TO S0 (begin);
S0: CASE(slot0_empty)
    1 => S1 (msg0 0,slot0_filled);
    0 => LOOP;
    ENDCASE => error(err);

S1: CASE(filled_ack0,ack_msg0 0,ack_msg1 0,ack_slot0_empty)
    1 1 0 0 => S2(ack_slot0_empty);
    1 0 1 0 => S3(ack_slot0_empty);
    0 0 0 1 => S4(ack_slot0_empty);
    0 0 0 0 => LOOP;
    ENDCASE => error(err);

S2: GOTO S5;
S3: GOTO S6;

S4: CASE(slot1_empty)
    1 => S7 (msg0 1,slot1_filled);
    0 => LOOP;
    ENDCASE => error(err);

S5: CASE(slot1_empty)
    1 => S8 (msg1 1,slot1_filled);
    0 => LOOP;
    ENDCASE => error(err);

S6: CASE(filled_ack1,ack_msg0 1,ack_msg1 1,ack_slot1_empty)
    1 1 0 0 => S9(ack_slot1_empty);
    1 0 1 0 => S10(ack_slot1_empty);
    0 0 0 1 => S11(ack_slot1_empty);
    0 0 0 0 => LOOP;

S7: CASE(filled_ack0,ack_msg0 0,ack_msg1 1)
    1 1 0 => S12(ack_slot0_empty);
    1 0 1 => S13(ack_slot0_empty);
    0 0 0 => LOOP;

S8: CASE(filled_ack1,ack_msg0 1,ack_msg1 1,ack_slot1_empty)
    1 1 0 0 => S14(ack_slot1_empty);
    1 0 1 0 => S15(ack_slot1_empty);
    0 0 0 1 => S16(ack_slot1_empty);
    0 0 0 0 => LOOP;

S9: GOTO S17;
S10: GOTO S1;
S11: CASE(slot1_empty)
    1 => S18 (msg0 1,slot1_filled);
    0 => LOOP;
    ENDCASE => error(err);

S12: GOTO S16;
S13: GOTO S18;
S14: GOTO S19;
S15: GOTO S0;
S16:  CASE(slot0_empty)
    1  =>  S20 (msgl_0,slot0_filled);
    0  =>  LOOP;
    ENDCASE  =>  error(err);

S17:  CASE(slot1_empty)
    1  =>  S19 (msgl_1,slot1_filled);
    0  =>  LOOP;
    ENDCASE  =>  error(err);

S18:  CASE(filled_ack1,ack_msg0_1,ack_msg1_1,ack_slot0_empty)
    I 1 0 0  =>  S21 (ack_slot1_empty);
    1 0 1 0  =>  S22 (ack_slot1_empty);
    0 0 0 1  =>  S23 (ack_slot1_empty);
    0 0 0 0  =>  LOOP;

S19:  CASE(filled_ack0,ack_msg0_0,ack_msg1_0,ack_slot0_empty)
    I 1 0 0  =>  S24 (ack_slot0_empty);
    1 0 1 0  =>  S25 (ack_slot0_empty);
    0 0 0 1  =>  S26 (ack_slot0_empty);
    0 0 0 0  =>  LOOP;

S20:  CASE(filled_ack1,ack_msg0_1,ack_msg1_1)
    I 1 0  =>  S27 (ack_slot1_empty);
    1 0 1  =>  S28 (ack_slot1_empty);
    0 0 0  =>  LOOP;

S21:  GOTO S26;

S22:  GOTO S7;

S23:  CASE(slot0_empty)
    1  =>  S6 (msg0_0,slot0_filled);
    0  =>  LOOP;
    ENDCASE  =>  error(err);

S24:  GOTO S8;

S25:  GOTO S23;

S26:  CASE(slot0_empty)
    1  =>  S29 (msgl_0,slot0_filled);
    0  =>  LOOP;
    ENDCASE  =>  error(err);

S27:  GOTO S29;

S28:  GOTO S4;

S29:  CASE(filled_ack0,ack_msg0_0,ack_msg1_0)
    I 1 0  =>  S30 (ack_slot0_empty);
    1 0 1  =>  S31 (ack_slot0_empty);
    0 0 0  =>  LOOP;

S30:  GOTO S20;

S31:  GOTO S11;

error: GOTO LOOP(err);

Figure 5.25: The meg input for Petri net model of sender of Alternating bit protocol
Figure 5.26: The layout of the sender of Petri net model of sender of Alternating bit protocol. Scalable cmos technology is used with cis style.
Chapter 6

Implementation of Concurrent systems from their Petri Net models

The previous chapter considered the problem of implementation of protocol controllers as a case study of concurrent systems. The first step in the implementation procedure was decomposition of the composite net (modelling both the entities) into subnets representing each entity. Three cases were enumerated, the first two imposed strong restrictions on the composite model. However, in the third case almost no restriction was imposed (except that it is amenable to hardware implementation). This situation occurs not only in communication protocols but in all concurrent/distributed systems. Distributed systems are a special case of concurrent systems where there is physical distribution of concurrent components. A distributed system is inherently concurrent (refer to the definition of distribution in [Jr78]).
With increasing use of concurrency in hardware design – multiprocessor systems (e.g. controllers) modern processors can be considered as cooperating processes. The design of such systems poses a problem. It was seen in the last chapter that synthesis using state machines does not guarantee that the overall system will be free of deadlock. The race for faster computer systems is bound to increase the use of concurrent systems. To quote from a recent work by Steve M. Nowick and David L. Dill [Men91b]:

Correct state machine synthesis cannot prevent bugs (synchronization problems) that occur from the interaction among several state machines. Synthesis from a high level concurrent program is desirable in some cases but the problem of how to divide a large computation into cooperating controllers is probably too difficult to automate.

They built finite state verifiers based on the trace theory of finite automata as an automatic verification tool. Their tool had exponential computational complexity. In this chapter the problem of decomposing a system (given by its Petri net model) into concurrent components is investigated. A procedure for this task is presented along with a discussion on its complexity. A special case of the procedure is then applied to case (2) of the decomposition treated in chapter 5.

6.1 Background and Definitions

To begin with a definition of concurrency is presented on the same lines as the work by Petri [Pet79]. Petri defined axioms for concurrency for occurrence nets
within the framework of condition/event modelling of systems, however, it can be used elsewhere too.

Concurrency can be regarded as a \textit{irreflexive}, \textit{symmetric} and \textit{non-transitive} i.e. a symmetric binary relation. Two events \( e_1 \) and \( e_2 \) are said to be concurrent if they can occur together. To illustrate this and the concurrency relation let us consider three events, \( e_1, e_2 \) and \( e_3 \) as singing, walking and driving respectively. Singing and walking are concurrent as they can be done at the same time. Then walking and singing too are concurrent (symmetry). Singing and driving too are concurrent. However, walking and driving are not (non-transitive). The reflexivity has been handled differently in the literature [Pet79,Pri82]. Formally, concurrency is defined as:

\textbf{Definition 1.6} Concurrency is a irreflexive, symmetric and non-transitive binary relation, \( \rho \) i.e.:

\[(x, y) \in \rho \implies (y, z) \in \rho\]

\[(x, y) \in \rho \implies x \neq y\]

and \((x, y) \in \rho \land (y, z) \in \rho\) does not \(\implies (x, z) \in \rho\)

Regarding Petri nets, with the condition/event model of system, places represent conditions and transitions denote events. The firing rule of Petri net allows only one transition to fire at a time. If two transitions, \( t_1 \) and \( t_2 \), are enabled at the same time, only one of them will fire. However there is no deterministic way to choose which one of these will fire. This restriction on firing will lead to both possible sequences, \( t_1 \) firing first and then \( t_2 \) firing or \( t_2 \) firing first and \( t_1 \) following,
occurring in the reachability graph. This appears as a 2-cube pattern in the graph. This happens because with two transitions enabled and no deterministic choice on firing there is no ordering of the two events. Any of them may fire (this was discussed in the section on concurrency in chapter 4). This is illustrated by a simple example shown in Figure 6.1. The explanation for considering concurrency as an irreflexive relation is based on the reachability graph. For live and bounded Petri nets (this subclass is of interest for practical applications, especially, hardware implementation) a transition being concurrent with itself implies the occurrence of a 2-cube structure with all its sides labelled by the same transition. This is shown in Figure 6.2. Such a structure cannot occur in the reachability graph [Val81b]. The following result is cited as a proof:

Result 1 (Valk) The language of firing sequences for unlabeled nets corresponds to deterministic finite automaton as for a given state and a given transition, only one state transition is possible. The reachability graph is isomorphic to this finite state automaton [Val81b].

However, if labels are assigned to the transitions and different transitions can be assigned the same label (different transitions perform the same action) then two transitions with the same label can fire in a marking and the FA is non-deterministic. In such a situation a 2-cube structure as shown in Figure 6.2 can occur.
Figure 6.1: (a) A Petri net with concurrency (b) the reachability graph for the Petri net
6.2 Concurrency Patterns in Reachability Graph

As seen in the last section two concurrent transitions in the Petri net give rise to a 2-cube structure in the reachability graph. Various other combinations were investigated and the results are presented next.

Case 1 Two concurrent transitions exhibit a 2-cube structure in the reachability graph. This was shown in Figure 6.1.

Case 2 Three concurrent transitions appear as a 3-cube in the reachability graph.

There are six possible sequences for the occurrence of three transitions, \( t_1, t_2 \) and \( t_3 - t_1t_2t_3, t_1t_3t_2, t_2t_1t_3, t_2t_3t_1, t_3t_1t_2 \) and \( t_3t_2t_1 \). This is shown in the Figure 6.3.

There are six paths between the two edges of internal diagonal of the cube. Each path is of length three and there are eight states that lie on the cube as its vertices.
Figure 6.3: A 3-cube structure for 3 concurrent transitions
Figure 6.4: A 4-cube structure for four concurrent transitions

Case 3 A 4-cube appears in the reachability graph of a Petri net having four transitions in parallel. This is shown in Figure 6.4.

After these illustrations the general theorem is presented.

Theorem 4 A Petri net has \( n \) transitions in parallel iff for all reachable markings the transition firings manifest themselves as an \( n \)-cube with the \( n \) transitions as its sides.
Proof 1 The proof is straightforward with the above discussion. If in a Petri net, 
n transitions are simultaneously enabled in a marking then as there is no ordering 
on their occurrence all the possible permutations will occur in the reachability 
graph. For a set with n elements, all possible permutations are n!, each of length 
n without any repetition. In an n-cube, between the two endpoints of the internal 
diagonal there are 2^n points and n! paths each of length n.

This theorem provides a method for detecting concurrently fireable transitions in 
the Petri net.

6.3 Detection of Concurrent Components

The problem of partitioning a Petri net exhibiting concurrency into interacting 
concurrent components is complicated by the fact that concurrency is not an 
equivalence relation and hence does not define a partition on the set of transitions. The objective of partitioning is that events which can occur in parallel be 
assigned to different subsets (T decomposition) and then each of the subset can 
be implemented as an FSM.

However, the mere detection of certain transitions being in parallel is not of much 
help as transitivity does not hold. For instance, if it is found that t_1, t_2 are in par-
allel and t_2, t_3 are concurrent then it does not imply that t_1, t_3 are also concurrent 
and they cannot be put into separate subsets unless it is explicitly known that 
they are parallel. The situation is quite complicated when there are a large num-
ber of such relationships between transitions. To handle this by a graph, co_graph
can be defined. The nodes of this graph are transitions (events) that exhibit concurrency. Two transitions that are concurrent are joined by an edge. The graph is undirected as direction is not needed (symmetry property). There are no self loops in the graph (irreflexive property). The graph is simple as there will be no parallel edges too (a graph with no parallel edges or self-loop is termed a simple graph [ST81]). The problem of assigning parallel events to different subsets can be reformulated in terms of the co-graph. Those vertices (transitions) which are connected by an edge should be assigned to different subsets. Moreover, the target is to achieve maximal concurrency. The following theorem relates the maximal number of concurrent components to the clique number of the graph. But first, the clique number of a graph is defined:

**Definition 17** A clique or a complete subgraph of a graph G is a subgraph whose every pair of vertices are adjacent.

**Definition 18** The clique number of a graph G, $\omega(G)$, is the maximum order among the cliques of the graph G.

**Theorem 5** The maximal number of concurrent components of the co-graph is given by the clique number of the graph, $\omega(G)$.

**Proof 2** As concurrency is not a transitive relation, for n elements to be concurrent there should be an edge from each node to the other $n - 1$ nodes i.e. the $n$ nodes form a clique. The clique number of a graph gives the maximum number of vertices in a clique. For maximal parallelism, the number of concurrent components should equal the clique number with each member of the highest order clique assigned to different partitions.
This theorem gives a measure of the maximal concurrency of the system. Next
task is to partition the set of nodes of co-graph such that all transitions that are
concurrent are allotted different subsets. This problem is equivalent to the well
studied graph coloring problem.

Graph coloring problem has been one of the most active area of research in graph
theory. Bounds on the chromatic number of a graph have been found. The defi-
nitions presented are from [CL86].

The graph coloring (of vertices) assigns colors to vertices such that no two ver-
tices that are adjacent get the same color. In this case, the vertices of the graph
represent events. Events which are concurrent are joined by edges. Coloring will
assign different colors to vertices that are parallel events. The color classes induce
a partition on the set of concurrent events such that each partition consists of
events that are not parallel to each other and each class by itself is concurrent
with the other classes.

Definition 19 *n-coloring* of vertices of a graph is one which uses *n* colors.

Definition 20 A graph, *G*, is *n-colorable* if there exists an *m*-coloring of *G* for
some *m* ≤ *n*.

Definition 21 The minimum number for which the graph is *n*-colorable is called
the *chromatic number* of the graph, χ(*G*) and the graph is said to be χ chromatic.

The chromatic number of a graph is related to the independent subsets of a graph.
Definition 22 An independent set of vertices of a graph $G$ is a subset of $G$ whose elements are pairwise non-adjacent.

Definition 23 The independence number, $\beta(G)$, of a graph is the maximum cardinality among the independent sets of the vertices of $G$.

Definition 24 The chromatic number of a graph is also defined as the minimum number of independent subsets in which $V(G)$ may be partitioned. Hence each independent set constitutes a color class.

Theorem 6 The $\chi(G)$-coloring of a graph gives the minimum color classes which are independent sets.

Proof 3 The proof comes from the definition of color classes.

With this the problem of detecting concurrent components can be reformulated as follows:

Problem Statement 1 Given a subset of transitions exhibiting concurrency, it is desired that it be partitioned into $n$ sets such that the maximal concurrency, $p$ is retained.

Obviously, $n \geq p$ with $n = p$ being the optimum. This can always be trivially done by increasing $n$. It is desired that the minimum number of components be used in order to achieve this. The outline of the algorithm is stated next:

1. Given a Petri net, $PN$. It is assumed that it is live and bounded as it is this class which is of interest for hardware implementation.

2. Find the reachability graph for the Petri net.
3. The existence of concurrent transitions can be detected by searching for n-cube patterns. An n-cube has the n transitions as its sides. It represents all the n! possible firing sequences of length n that can occur as paths between the endpoints of the internal diagonal and 2^n states lie on the vertices of the cube. If such a pattern occurs in the reachability graph then these n transitions are concurrent.

4. For the concurrent transitions build a simple, undirected graph, called concurrency graph, CoG, as:

\[ \text{CoG} = (V, E) \] where,

\[ V \subseteq T, \]

is the set of nodes of the graph with each node representing a transition which is concurrent to some other transition

\[ E \subseteq V \times V \]

is the set of edges, with

\[ e_{i,j} \in E \text{ iff node(transition) } i \text{ is concurrent with node(transition) } j.\]

5. The maximal parallelism is given by the clique number, \( \omega(\text{CoG}) \) of the graph.

6. The minimum number of independent subsets into which the graph can be partitioned is given by the \( \chi(\text{CoG}) \)-coloring of the graph.

7. The transitions that do not occur in parallel with any other transition and were not featured in the concurrency graph, CoG are necessarily sequential.
and are related to the concurrent by precedence relation. These can be assigned to the color class which contains the related transitions. This resolves the partitioning problem; the result is a partition with maximal concurrency.

6.4 Complexity of the Procedure

The procedure presented has very high computational complexity. The first step involves finding the reachability graph. For a Petri net, the reachability graph is an exhaustive procedure and requires space and time exponential in the number of places $|P|$.

The task of detection of n-cubes from the reachability graph can be done in polynomial time (it is shown in the next section that a modified shortest path detection algorithm based on breadth first technique can be used) but the graph is exponential in $|P|$.

The graph coloring problem has been shown to be NP-complete [HS78] and there is no $\epsilon$ approximation (an algorithm that gives the result which is close to the optimal and the error is bounded by $\epsilon$) algorithm in polynomial time too. The size of the CoG graph is $O(|T|)$ (the nodes of the graph are a subset of $T$) so the coloring problem is exponential in $|T|$.

In all, the procedure has exponential complexity. Although state space of concurrent systems becomes very large and reachability graphs have exponential complexity yet they remain the basic analysis technique. So the overhead involved in
adding concurrency detection to the analysis is not much. Alternatively, the de-
tection of concurrent transitions has to be done using structural analysis techniques.
This would eliminate the use of reachability graph and reduce the complexity. The
\( \chi(G) \)-coloring and clique number have been shown to be of exponential complex-
ity. However, this simply means that heuristics have to be developed based on the
particular application to obtain optimal solutions. Probabilistic algorithms such
as simulated annealing may be used [McH90]. A large number of applications use
heuristics to solve NP-complete problems in reasonable time.

6.5 Implementation Aspects

The implementation aspects of the various procedures are presented:

6.5.1 Detection of n-cube

The detection of n-cube can be reformulated as a shortest path problem and solved
using a modified Breadth First Search Technique [Eve80].

Procedure 5 (Detect_Cube)

Input The Reachability Graph

Output n-cubes

1. Do for all states, beginning from state 0 corresponding to \( M_0 \)

2. If at a state \( Sr \) there are \( n \) outgoing arcs labelled by transitions, \( t_1, t_2, \ldots, t_n \)
   then find whether they lie on an n-cube. This can be done by the following
   procedure:
3. assign each of these transitions a cost of 1 and all others infinity

4. find all paths of length $n$ from state, $S_r$

5. if all paths are of length $n$, terminate at vertex $T$ and there are $n!$ paths with vertices (including $S_r$ and $S_p$) lying on the path being $2^n$ then the transitions lie on an $n$-cube.

The Breadth First Search Algorithm is presented below –

Algorithm 1 (BFS_Cube)

begin
$S \leftarrow 0$
$i \leftarrow 0$
$nodes \leftarrow 0$

do 1 to $n$

if no_of_neighbours at distance of 1 $< (n-i)$ then

The transitions do not fall on an $n$-cube

return

else

nodes $\leftarrow$ nodes + $(n-i)$

label each next_neighbour as $i+1$

$i \leftarrow i + 1$

end do

return
Figure 6.5: Application of Breadth First Technique on a 3-cube structure

At the end of procedure the label of the terminal node Sp should be n and the value of variable nodes should equal $2^n$. An illustration of this figure for a 3-cube is shown in Figure 6.5

6.5.2 Clique number and Chromatic number

The maximal parallelism is related to the clique number of a graph. The chromatic number and the clique number are related too. Clique number is in fact a lower bound on the chromatic number. Two results on the lower bound are taken from
the literature [CL86, Rob84] and presented here:

Result 2 (Lower Bound) For every graph, \( \chi(G) \geq \omega(G) \)

This is a result of the pigeonhole principle. This lower bound is tight and achieved for perfect graphs.

Result 3 (Seinsche) If a graph does not contain \( P_4 \) as an induced subgraph then \( \chi(G) = \omega(G) \) where \( P_4 \) stands for a path of length 4 [Sei74].

A number of upper bounds have been proposed which vary in their estimates for different classes of graphs:

Result 4 (Brooks) If \( G \) is a connected graph that is neither an odd cycle nor a complete graph then, \( \chi(G) \leq \Delta(G) \) where \( \Delta(G) \) stands for the maximum degree of nodes of the graph (the maximum edges connected to a node) [Bro41].

A corollary of the above theorem is that –

Result 5 For every graph, \( G, \chi(G) \leq 1 + \Delta(G) \) where \( \Delta(G) \) stands for the maximum degree of nodes of the graph.

Other upper bounds include:

Result 6 For every graph, \( G, \chi(G) \leq 1 + \max(\delta(G)) \) where \( \delta(G) \) stands for the minimum degree of the graph and maximum is taken over all induced subgraphs of \( G \).

There is another interesting upper bound:

Result 7 If a graph is \( n \)-partite, \( \chi(G) \leq n \)
No efficient algorithms exist for graph coloring. An algorithm called sequential coloring algorithm defines another upper bound on the chromatic number.

**Algorithm 2 (Sequential Coloring)**

**Input** A graph with labelled vertices $V(G) = \{V_1, V_2, \ldots, V_p\}$

**Output** A colored graph

1. Let $i = 1$ /* $i$ is an index for nodes */
2. Let $c = 1$ /* $c$ is an index for colors */
3. if no vertex adjacent with $V_i$ is assigned color $c$ then assign $c$ to $V_i$
4. else replace $c$ by $c+1$
5. if $i < p$ then $i = i+1$ and Goto step 3
6. stop

**Result 8 (WELSH)** The sequential coloring algorithm defines an upper bound $\chi(G) \leq \max(\min(i, \deg V_i + 1))$, the maximum is taken for $1 \leq i \leq p$ [WP67]

The coloring is a function of the labels assigned to the nodes.

These bounds give an idea of the optimal results. The upper bound is easy to calculate although it might not be in all types of graph close to the actual. A heuristic coloring algorithm was proposed by [KP84] which readily yields a result. This can be used to generate a coloring which might not be close to optimal.
Reduction of size of co-graph

As the chromatic number problem is NP-complete a reduction in the size of the co-graph would greatly reduce the complexity of the problem. A heuristic method of reducing the size of the co-graph is presented.

In practice concurrent/distributed systems are basically comprised of cooperating sequential processes. Each sequential process is autonomous except for message exchanges with other processes for synchronization to achieve an overall behavior. Moreover these processes definitely communicate with their environment. Hence events within a process can be classified into two types –

1. actions through which the process communicates with the environment and

2. local actions.

the local actions do not depend on other processes and hence they should appear concurrent with the local actions of other processes. Moreover, since the processes are autonomous except for limited exchange it seems improbable that the reachability graph will exhibit n-cube type patterns, where single transitions occur in parallel. In fact, fairly autonomous processes will have sequence of transitions in parallel. This is illustrated in Figure 6.6 where two sequences of different lengths are shown in parallel. For two parallel sequences of length 1 yield a single 2-cube, length 2 yields 4 adjacent 2-cubes and length 3 yields 9 adjacent 2-cubes. This can be explained by the fact that for two single transitions, \(x||a\) (\(||\) denotes parallel), 2! permutations are possible and a 2-cube is obtained. When the length is made two, \(xy||ab\) then \(x||a \& x||b\) and \(y||a \& y||b\) so each transition is parallel to the
Figure 6.6: (a) $S_1 S_2$ (b) $S_1 S_2 || V_1 V_2$ and (c) $S_1 S_2 S_3 || V_1 V_2 V_3$
entire length of other sequence and the number of 2-cubes will be \((\text{length})^2\). This can be generalized by the following theorem:

**Theorem 7** For \(n\) sequences of length \(m_1, m_2, \ldots, m_n\) the pattern obtained will have \(m_1 \times m_2 \times \cdots \times m_n\) \(n\)-cubes.

**Proof 4** The proof can be given along the same lines as the above discussion.

This can be used to reduce the size of the co-graph by replacing each sequence by a single node. The detection of sequences can be heuristically done as presented below. This can in place of the cube detection procedure.

**Procedure 6 (Sequence Detection)**

**Input** The reachability graph

**Output** Sequence of parallel events

1. The cube detection can be done by the routine described previously

2. Starting from the initial marking look for an \(n\)-cube

3. if an \(n\)-cube is detected with entry state, \(S_r\) and terminating state, \(S_p\).

4. for all the successors of \(S_r\) (Breadth First Search) try to find if another \(m\)-cube exists

5. If a cube is found for the successor, \(S_{r'}\), store the transition labelling the incoming arcs to \(S_{r'}\) and already detected in a previous cube as predecessors for the cube rooted at \(S_{r'}\) and the node \(S_{r'}\) in the list for the next stage \(S_r\).

6. find the cube rooted at \(S_{r'}\)
7. repeat until the next stage Sr list is empty

8. from the cubes detected, gather the information about all the transitions to which each transition is parallel to and the predecessors of each transition (is the predecessor of the root of each cube for which the transition was a part).

9. Create a table with one entry per transition. It has two fields, one for the predecessors of the transition and the other for transitions to which it is parallel.

10. delete transitions that are common to both the predecessor and parallel transitions from the predecessor list of each transition. This is because for concurrent transitions the order is irrelevant.

11. pick a transition form a sequence by catenating the transition for which it appears as a predecessor. This is repeated until the sequence no more grows.

12. this is repeated till all transitions are accounted for.

13. merge the transitions that are parallel to a sequence, replace those which form a sequence by a single entry.

14. build the co.graph for the sequences rather than individual transitions.

This is illustrated by an example (Figure 6.7) showing the main steps. The size of the concurrency graph can be further reduced by using some of the results on chromatic number of a graph. Before the results, the terms used are defined.
Figure 6.7: Illustration of heuristic for detecting sequences (a) The concurrency pattern (b) table showing the transitions, their predecessors and parallel transitions and (c) sequences formed
Definition 25 A vertex, $v$, of a connected graph is a cut vertex iff the graph $G-v$ (obtained by deleting the vertex $v$ and the edges connected to it) is disconnected into at least two components. A complete graph has no cut vertices and a graph with no cut-vertices is called non-separable.

Definition 26 A block of a separable graph is a maximal non-separable subgraph of $G$.

Now the results are presented, these appear in [CL86].

Result 9 The chromatic number of a disconnected graph is the maximum of the chromatic numbers of its components.

Result 10 The chromatic number of a connected graph with cut-vertices is the maximum of the chromatic numbers of its blocks.

The blocks of a graph can be found in polynomial time using a depth first search tree. An algorithm is presented in [McH90]. The procedure for finding the $\chi$(CoG)-coloring is summarized below:

1. Build the co-graph using sequences of parallel transitions

2. find the blocks of the CoG using the algorithm for block detection

3. for each block, find the chromatic number coloring of the CoG using some heuristic algorithm

4. The chromatic number of CoG is the maximum of it's block's chromatic numbers and defines the color classes.
A heuristic algorithm for $\chi(G)$-coloring was given by [KF84]. The above procedure is generalized and is to be applied when only the composite model is known and the information about the number of components comprising the process and the processes are to be found from the graph. By putting restrictions on the this general case computationally less expensive procedures can be obtained. For instance, if the number of components is known (as in case of controllers for databases), a backtracking $m$-coloring algorithm can be used [HS78].

6.6 Special Case

Now after considering the general case for detection of interacting components a special case is considered. This is the case (2) that was mentioned in the chapter 5. The lower layer communication protocols, e.g., data link layer are suitable for hardware implementation. For a 2-party, sender receiver protocol the graph coloring problem reduces from finding $n$ concurrent components to finding two components. Hence the problem of finding transitions that belong to different processes in a protocol is equivalent to finding a 2-coloring of the co-graph. This can be done in polynomial time by using the result of the following theorem from [CL86].

Theorem 8 If $G$ is a 2 chromatic graph then necessarily $G$ is bipartite and the color classes obtained by any 2-coloring of $G$ are partite sets of the bipartite graphs.

The detection of bipartite components of a graph can be done in polynomial time by a depth first search algorithm [McH90] (the complexity is $O(|V| + |E|)$). This special case is illustrated by an example shown in Figure 6.8. However there is a
Figure 6.8: (a) The Petri net model (b) the reachability graph (c) the CoG graph with the partite components shown.
catch that the process at each site must have at least one local action otherwise it may not exhibit concurrency. It also might happen that some of the transitions do not occur in the CoG graph as they are not concurrent with any other transition (might occur if the merged transition technique is used). They must then appear as sequential event with some other process. This can be found by cutting all the arcs to a transitions if there are more than one of them coming in or going out. This leaves the state machine processes connected. For the transitions that do not appear in CoG find their predecessor and successor. Once this is done for all such transitions then they can be assigned to one of the color classes which has a successor or predecessor of that transition. If this cannot be done then they can be distributed among the color classes so as to, for instance, make the number of elements of the classes equal. An example shown in Figure 6.9 is used to illustrate this. this would then define a partition on the set of transitions. For shared place interconnection mechanism the situation is similar to case (0) discussed in chapter 5. However if there is no information on the composition technique then an automatic separation of the global FSA into distributed components can be done using the algorithm proposed by [Pri82]. This algorithm accepts a FSA and a partition on the input alphabet such that concurrent events are assigned different to different sets and outputs a distributed FSA with each partition defining a FSA and channels are generated for communication between the processes. In this case the reachability graph is the global FSA and the transitions labelling the arcs are the input alphabet. Once the output of the C algorithm is obtained the situation becomes similar to the case (0) and the same sequence of steps can be applied to generate the VLSI layout of the PLA automatically.
Figure 6.9: (a) The Petri net model (b) the reachability graph and (c) the transitions not appearing in concurrency graph.
Chapter 7

Conclusions and Future Work

The implementation of concurrent systems from their Petri net model was investigated. Communication Protocols were studied as an instance of concurrent/distributed systems. The following things can be concluded from the study:

1. Communication protocols, in the most general case, when no information is available about the system, can be implemented using the algorithm presented. This algorithm has exponential complexity which arises from the use of reachability graph. Although reachability graph suffers from state explosion, especially for concurrent systems it is still used to verify properties and moderate sized protocols it can be handled. The detection of concurrent components is reduced to detection of bipartite components which can be done in polynomial time. The overall complexity of implementation will be exponential in the number of places in the Petri net model.

2. If the protocol model is restricted to deterministic system of sequential processes then the task of detection of the components can be done using struc-
tural analysis and this would have polynomial complexity. Given that they have been shown to be sufficient to model protocols, it seems attractive to put some restriction on the model designer. Moreover, this has advantages in the analysis phase of the design too. The liveness and boundedness properties of the composite net are related to that of component nets and hence presents a big saving in computational effort.

3. If the model of the protocol is known with the entities demarcated then the implementation is easy and can be easily automated using the algorithms presented.

4. In the general case of concurrent systems, it was seen that the task of finding the minimum number of components that retain the maximal concurrency of the Petri net model involves $\chi(\text{CoG})$-coloring. The overall complexity is exponential. Here again reachability proves to be the bottleneck. Heuristics were presented to reduce the size of the CoG graph and heuristic algorithms (including probabilistic algorithms such as simulated annealing) can be used for the coloring problem.

5. If the number of components is known (e. g., in database controller synthesis) then a backtracking algorithm can be used to find a m-coloring [HS78].

In view of the above comments the following areas need further investigation:

- The main bottleneck is the reachability graph. It should be investigated how concurrent transitions can be found without using the reachability graph, i.e. from the structural analysis. For instance, from the incidence matrix.

- Efficient heuristics for coloring have to be investigated.
• Once concurrent components are found, some of the transitions might not appear in the concurrency graph. Algorithm for detecting these and their precedence relations automatically have to be found.

• An alternative to using structural properties for detecting concurrency is to find how the concurrency detection can be mapped to high level Petri nets (Predicate/Transition nets, Colored Petri nets). The high level Petri nets do not suffer from state space explosion problem as the ordinary Petri nets.
Bibliography


[MBM89] Teresa H. - Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high level


Vita

Asjad Mumtaz Tahir Khan

Born at Aligarh, India

Received Bachelor’s degree in Electrical Engineering from the Aligarh Muslim University, Aligarh, India in July 1988.