INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

GRADUATE PROGRAM IN INFORMATION TECHNOLOGY
AND ELECTRONICS.

GENERIC AND MODULAR MODEL TO DEVELOP VIRTUAL
LABORATORIES FOR TELEROBOTICS OVER THE INTERNET

THESIS

PRESENTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE ACADEMIC DEGREE OF:

DOCTOR OF PHILOSOPHY IN
ARTIFICIAL INTELLIGENCE

BY

FERNANDO DANIEL VON BORSTEL LUNA

Monterrey, N.L. November 2005
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

GRADUATE PROGRAM IN INFORMATION TECHNOLOGY AND ELECTRONICS

GENERIC AND MODULAR MODEL TO DEVELOP VIRTUAL LABORATORIES FOR TELEROBOTICS OVER THE INTERNET

THESIS

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE ACADEMIC DEGREE OF:

DOCTOR OF PHILOSOPHY IN ARTIFICIAL INTELLIGENCE

BY

FERNANDO DANIEL VON BORSTEL LUNA

Monterrey, N.L. November 2005
SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Fernando Daniel Von Borstel Luna

This dissertation has been read by each member of the following supervisory committee, and by majority vote has been found to be satisfactory.

Supervisory Committee

José Luis Gordillo Moscoso Ph.D.
Advisor

Enrique Sucar Succar, Ph.D.
Member

Ciro A. Rodríguez González, Ph.D.
Member

I. Barrón Cano, Ph.D.
Member and Committee Chair

Manuel E. Macías García, Ph.D.
Member

Monterrey N. L., November 2005
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

FINAL READING APPROVAL

To the Graduate Program of Instituto Tecnológico y de Estudios Superiores de Monterrey:

I have read the dissertation of Fernando Daniel Von Borstel Luna in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative material including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to the Graduate Program.

Date

José Luis Gordillo Moscoso, Ph.D.
Advisor

Approved by the Program Coordinator

Hugo Terashima Marín, Ph.D.
Program Coordinator

Approved by the Graduate Council

David Garza Salazar, Ph.D.
Dean of the Graduate Program
Dedication

This thesis is dedicated to my wife Obsidiana, my daughter Sarahi, my son David, my parents Nely and Raul, my brothers and sisters, and my in-laws whose guidance, support, love, and enthusiasm encouraged me to reach my objective.
Acknowledgments

I express my gratitude to my thesis advisor, Dr. José Luis Gordillo, whose expertise, understanding, and patience contributed considerably to my graduate experience. I appreciate his vast knowledge and skills in many areas, as well as his perceptive comments that contributed to the shape and clarity of my original manuscript.

I acknowledge other members of my thesis committee: Dr. Olivia Barrón, Dr. Manuel Macías, and Dr. Ciro Rodríguez from Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM) Campus Monterrey, and Dr. Enrique Sucar from ITESM Campus Cuernavaca, for taking time out from their busy schedules to serve as my committee members. Their insightful comments and corrections helped me to improve the quality of my research in all the topics which are included in this thesis.

I wish to thank all my teachers in Centro de Sistemas Inteligentes (CSI), in particular Dr. Rogelio Soto, Director of CSI and Dr. Hugo Terashima, Coordinator of the Graduate Program in Artificial Intelligence, for their guidance and support during my entire Ph.D. study, and for being available whenever I needed an advice.

My special thanks to:

My patient and loving wife Obsidiana, my children Sarahi and David, my parents Nely and Raul, my brothers and sisters, and my in-laws, whose love, encouragement, and support, helped me to finish this thesis.

Members of Grupo Visión in CSI: Carlitos, Joaquin, Brian, Hugo, Gerardo, Patricia, and Miguel Angel, for their patience, exchanges of knowledge, skills, and moral support.

My friends and classmates: Doris, Luis Jorge, German, Salvador, Hector, Griselda, Victor, Eduardo, Roberto, Rolando, Cesar, and many others, for their optimism and encouragement.

My colleagues and friends in Centro de Investigaciones Biológicas del Noroeste, S.C. (CIBNOR): Elizabeth, Driscema, Laura, Hector, Roberto, Horacio, Isabel, Pedro, Oscar, for their support and friendship.

Diana Fischer and Ira Fogel, for their help in correcting and improving my English writing.

Finally, I recognize that this research would not have been possible without the support of ITESM, Consejo Nacional de Ciencia y Tecnología, and CIBNOR to whom I express my gratitude.
Abstract

Sustained advances in information and communication technologies have made it possible to access remote and globally distributed resources for instruction, information and group collaboration over the Internet. In this context, the Virtual Laboratory (VL) concept emerges to offer heterogeneous and distributed environments to perform remote experimentation by operating simulated equipment or remotely operating real equipment. Despite its importance in educational and research fields, VLs are not developed in a systematic way, since its implementation integrates multiple technologies to communicate through the Internet, control heterogeneous equipment, and simulate experiments, among others; VLs are designed and implemented based on the experience of the developer, using intuitive and tested approaches, disregarding what functionalities are needed to perform experimentation, creating partial models that ignore structural composition or behavioral design, using ad hoc architectures or development frameworks that depend on new features of interconnection technologies.

Research performed in this work attempts to bridge the gap between intuitive methodologies and a formal and general methodology to develop VLs for telerobotics over the Internet. The performed research creates a generic and modular VL model with three essential elements: Guest, Media, and Host. The proposed model allows the developer to be aware of tested teleoperation strategies. The model is composed of generic entities, which are vendor and technology independent to define the appropriate configuration of desired computer-based processes in a VL for telerobotics. This model is related to the event-based control theory through a teleoperation architecture to introduce control design requirements for stable and synchronized Internet-based VL applications. The model is transformed into a reference framework, which can be customized, based on the experiment specifications. Three functionalities are defined in the reference framework to allow the user to remotely perform a true experiment: mount, define, and execute. The unified modeling language (UML) is used to describe in detail the structure and dynamics of these functionalities based on the object-oriented paradigm.

The object-oriented reference framework is taken as the starting point by a general methodology to avoid costly development of new VL applications from scratch. This development methodology has four phases. The first phase analyzes the experiment and defines its functionalities and their inherent components. The second phase identifies and instantiates components into the UML framework. The customized framework describes experiment functionalities as VL subsystems. In the third phase, a novel procedure formally translates the dynamics of the customized framework into the Petri Net notation to carry out a quantitative and qualitative analysis, which relates the Petri Net design with event-based control properties. The fourth phase merges VL subsystems to compose a complete Petri Net design, which is analyzed to validate the sequential execution of VL subsystems. If necessary, the developer can synthesize Petri Net structures to control its behavior. This phase verifies that
experiment functionalities perform sequentially, assuring a true experimentation, and validates that the dynamic design represents a stable event-based system over the Internet. The generated structural and dynamic designs provided by UML and Petri Net models supply a guideline to the developer to implement a VL, based on the object-oriented paradigm and the event-based control. Generated UML diagrams allow producing, high quality and low cost software artifacts. Petri Net diagrams provide a comprehensive design of the inherent control of software components. This development methodology proposes an original approach for modeling, designing, analyzing, and validating the structure and dynamics of VL applications for telerobotics.

To conclude, two VLs for telerobotics are developed in this study, using the proposed methodology. A VL for mobile robotics is designed and implemented. It uses potential field and computer vision techniques to remotely plan and follow robot trajectories. After that, a VL for bilateral teleoperation that provides visual and haptic feedback is also designed and implemented. It uses a real-time event-based controller to provide sensory information. This VL application reuses software artifacts of the previously designed VL. Experiments are carried out via Internet and Internet2 (Media), using test beds that allow remote interactions between Guest and Host that are geographically separated. All the experimental results confirm the presented theory.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>xvii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xxiii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xxvii</td>
</tr>
</tbody>
</table>

## 1 Introduction
1.1 Problem Statement .................................. 3  
1.2 Thesis Statement .................................... 3  
1.3 Methodology ........................................... 4  
1.4 Scope of the Thesis .................................. 8  
1.5 Dissertation Overview ................................ 9  

## 2 Overview of Previous Work
2.1 The Internet as a Communication Medium .................. 12  
2.2 Time-delayed Teleoperation .............................. 13  
2.2.1 Direct Teleoperation ................................ 13  
2.2.2 Indirect Teleoperation ............................... 14  
2.3 Bilateral Teleoperation ................................ 15  
2.4 Distributed Telerobotics ............................... 16  
2.5 Virtual Laboratories ................................ 17  
2.6 Limitations of Existing Works .......................... 18  
2.7 Overview Conclusion ................................... 19  

## 3 Virtual Laboratory Modeling
3.1 Virtual Laboratory Concept ................................ 21  
3.2 Model Elements ......................................... 24  
3.3 Abstract Model ......................................... 25  
3.4 Object-Oriented Modeling ................................ 26  
3.4.1 Virtual Laboratory Classes ........................... 28  
3.4.2 Simulated Virtual Laboratory .......................... 29  
3.4.3 Remote Virtual Laboratory ............................ 30  
3.4.4 *Guest* and *Host* Software .......................... 31  
3.4.5 Software Collaboration Diagram ....................... 32  
3.4.6 Deployment Diagram .................................. 32  

xvii
## CONTENTS

3.4.7 Use Case Diagram ................................................. 33
3.4.8 Sequence Diagram .................................................. 35
3.4.9 Statechart Diagram .................................................. 35
3.4.10 UML Statechart Diagram Formal Definition .................. 38
3.5 Event-based Architecture ............................................. 39
  3.5.1 Event-Based Planning and Control Theory ...................... 39
  3.5.2 Event-Based Design Requirements .............................. 42
  3.5.3 Petri Net Formal Definition ..................................... 42
  3.5.4 Proposed Architecture ........................................... 44
3.6 VL Modeling Conclusion ............................................. 49

4 Development Methodology ............................................. 53
  4.1 Experiment Design and Functionality Analysis .................... 55
  4.2 Subsystem Identification in the VL Framework .................... 55
  4.3 Conversion into Dynamic Model and Analysis ...................... 57
    4.3.1 Statechart diagram-Petri Net Conversion .................... 59
    4.3.2 Petri Net Analysis ............................................. 63
    4.3.3 Event-Based VL Subsystem Analysis Using Petri Nets ........ 64
  4.4 Composition, Analysis and Control of a Dynamic Model .......... 65
    4.4.1 Synchronous Transition Composition Definition ............... 66
    4.4.2 Parallel Place Fusion Definition ............................... 67
    4.4.3 Behavioral Analysis of Composed Petri Net .................... 68
    4.4.4 Control Design for Composed Petri Net ....................... 69
    4.4.5 Supervisory Control of DES using Petri Nets ................... 69
    4.4.6 Constraint Forms to Synthesize Controller Nets ............... 70
    4.4.7 Logical Control Method for a DES ............................. 71
    4.4.8 Behavioral Analysis of Controlled Petri Net .................... 74
  4.5 Structural and Dynamic Designs Implementation .................. 75
  4.6 Advantages of the Proposed Methodology .......................... 75
  4.7 Conclusions ...................................................... 77

5 Design of a Virtual Laboratory for Mobile Robotics ............... 79
  5.1 Experiment Design and Functionality Analysis .................... 81
    5.1.1 Experiment Description ........................................ 81
    5.1.2 Experiment Analysis .......................................... 83
  5.2 C-Space Creation Functionality Modeling .......................... 84
    5.2.1 C-Space Creation Functionality Analysis ..................... 84
    5.2.2 C-Space Creation Subsystem Identification .................... 85
    5.2.3 UML-Petri Net Conversion and Analysis ....................... 90
  5.3 Path Planning and Path Following Functionalities Modeling ....... 93
  5.4 Petri Net Synchronous Composition .................................. 93
    5.4.1 Composed Petri Net Analysis .................................. 97
    5.4.2 Petri Net Control Design ...................................... 98
    5.4.3 Final Dynamic Analysis ....................................... 103
5.5 VL System Implementation .............................................. 105
5.6 VL Development Conclusions ......................................... 105

6 Design of a Virtual Laboratory for Bilateral Teleoperation .......................... 107
6.1 The Teleoperation Experiment ......................................... 107
6.2 Real-Time Control ....................................................... 108
   6.2.1 Control Design .................................................. 109
   6.2.2 Experiment Processes .......................................... 113
   6.2.3 Experiment Functionality Analysis ......................... 114
6.3 C-Space Creation Functionality Modeling ................................ 115
6.4 Potential Field Creation Functionality Modeling ....................... 116
   6.4.1 Analysis of the Potential Field Creation Functionality ...... 116
   6.4.2 Potential Field Creation Subsystem Identification .......... 116
   6.4.3 UML-Petri Net Conversion and Analysis ..................... 119
6.5 Robot Teleoperation Functionality Modeling ........................... 122
6.6 Petri Net Synchronous Composition ................................... 122
   6.6.1 Composed Petri Net Analysis ................................ 126
6.7 Petri Net Control ....................................................... 127
   6.7.1 Final Dynamic Analysis ....................................... 129
6.8 VL System Implementation ............................................. 131
6.9 VL Development Conclusions ......................................... 131

7 Implementation and Experimental Results .................................... 133
7.1 Hardware Description and Specifications ................................ 133
7.2 Software Description and Specifications ............................... 134
7.3 Guest and Host Geographic Location and Media Specifications ........... 136
7.4 Remote Experimentation Using the VL for Mobile Robotics ............. 137
   7.4.1 System Specifications ......................................... 137
   7.4.2 Path Analysis .................................................. 139
   7.4.3 Time Analysis .................................................. 140
7.5 Experimentation Using the VL for Bilateral Teleoperation ............... 142
   7.5.1 System Specifications ......................................... 142
   7.5.2 Experimental Results .......................................... 143
7.6 Implementation and Experimentation Concluding Remarks ................ 146

8 Conclusions .................................................................... 147
8.1 Contributions ............................................................. 148
8.2 Future Directions ......................................................... 149

A UML Software Component Definition ........................................ 151
A.1 Guest Software .......................................................... 151
   A.1.1 Input Processing .................................................. 151
   A.1.2 Simulation .......................................................... 152
   A.1.3 Sensory Conversion .............................................. 154
   A.1.4 Settings/Tasks Database ......................................... 155
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.2</td>
<td>Host Software</td>
<td>156</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Sensed Data Processing</td>
<td>156</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Experiment Control</td>
<td>157</td>
</tr>
<tr>
<td>A.2.3</td>
<td>Model Libraries</td>
<td>158</td>
</tr>
<tr>
<td>A.2.4</td>
<td>Experiment Database</td>
<td>159</td>
</tr>
<tr>
<td>A.3</td>
<td>Communication</td>
<td>159</td>
</tr>
<tr>
<td>B</td>
<td>VL for Mobile Robotics: Robotics and Vision Techniques</td>
<td>163</td>
</tr>
<tr>
<td>B.1</td>
<td>C-Space Creation</td>
<td>163</td>
</tr>
<tr>
<td>B.1.1</td>
<td>C-Space Definition</td>
<td>163</td>
</tr>
<tr>
<td>B.1.2</td>
<td>C-space Computation</td>
<td>164</td>
</tr>
<tr>
<td>B.2</td>
<td>Path Planning</td>
<td>166</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Simple Navigation Function NF1</td>
<td>166</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Gradient Descent Algorithm</td>
<td>168</td>
</tr>
<tr>
<td>B.2.3</td>
<td>Curve Interpolation</td>
<td>169</td>
</tr>
<tr>
<td>B.3</td>
<td>Path Following</td>
<td>171</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Image Definition and Binarization</td>
<td>172</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Image Segmentation</td>
<td>172</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Object Characterization</td>
<td>175</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Robot Control</td>
<td>176</td>
</tr>
<tr>
<td>C</td>
<td>VL for Bilateral Teleoperation: Robotics and Vision Techniques</td>
<td>179</td>
</tr>
<tr>
<td>C.1</td>
<td>Robot Position Prediction</td>
<td>179</td>
</tr>
<tr>
<td>C.1.1</td>
<td>Differential Drive</td>
<td>179</td>
</tr>
<tr>
<td>C.1.2</td>
<td>Forward Kinematics</td>
<td>181</td>
</tr>
<tr>
<td>C.1.3</td>
<td>Khepera Characterization</td>
<td>182</td>
</tr>
<tr>
<td>C.2</td>
<td>Khepera Speed Controller Characterization</td>
<td>183</td>
</tr>
<tr>
<td>C.3</td>
<td>Potential Field Creation</td>
<td>184</td>
</tr>
<tr>
<td>C.4</td>
<td>Image Data Compression</td>
<td>186</td>
</tr>
<tr>
<td>D</td>
<td>Functionality Modeling Phases</td>
<td>189</td>
</tr>
<tr>
<td>D.1</td>
<td>VL for Mobile Robotics Functionality Modeling</td>
<td>189</td>
</tr>
<tr>
<td>D.1.1</td>
<td>Path Planning Functionality Modeling</td>
<td>189</td>
</tr>
<tr>
<td>D.1.2</td>
<td>Path Planning Functionality Analysis</td>
<td>190</td>
</tr>
<tr>
<td>D.1.3</td>
<td>Path Planning Subsystem Identification in the VL Framework</td>
<td>190</td>
</tr>
<tr>
<td>D.1.4</td>
<td>UML-Petri Net Conversion and Analysis</td>
<td>194</td>
</tr>
<tr>
<td>D.1.5</td>
<td>Path Following Functionality Modeling</td>
<td>195</td>
</tr>
<tr>
<td>D.1.6</td>
<td>Path Following Functionality Analysis</td>
<td>196</td>
</tr>
<tr>
<td>D.1.7</td>
<td>Path Following Subsystem Identification in the VL Framework</td>
<td>196</td>
</tr>
<tr>
<td>D.1.8</td>
<td>UML-Petri Net Conversion and Analysis</td>
<td>199</td>
</tr>
<tr>
<td>D.2</td>
<td>VL for Bilateral Teleoperation Functionality Modeling</td>
<td>202</td>
</tr>
<tr>
<td>D.2.1</td>
<td>Robot Teleoperation Functionality Modeling</td>
<td>202</td>
</tr>
<tr>
<td>D.2.2</td>
<td>Robot Teleoperation Subsystem Identification</td>
<td>202</td>
</tr>
<tr>
<td>D.2.3</td>
<td>UML-Petri Net Conversion and Analysis</td>
<td>203</td>
</tr>
</tbody>
</table>
E Observer System 209

F VL for Mobile Robotics Improvements 211
  F.1 Using Robot Sensors to Avoid Physical Obstacles 212
  F.2 Virtual Obstacles Insertion 213
  F.3 Avoiding Collisions with Virtual Obstacles 214
  F.4 Modifying the Planned Path 215
  F.5 Robot Teleoperation 215
  F.6 Adjustable Settings 216

G Decreasing the Number of Transmitted Images 219
  G.1 Implemented Method to Decrease the Number of Transmitted Images 219
  G.2 Dynamic Models for both VLs 221

Bibliography 225
# List of Figures

1.1 Virtual Laboratory model ........................................... 6
1.2 VL development methodology ..................................... 7

3.1 Simulated and remote Virtual Laboratories ...................... 22
3.2 Virtual Laboratory elements ....................................... 24
3.3 Abstract VL model for telerobotics ................................. 25
3.4 Virtual Laboratory classes .......................................... 28
3.5 Simulated Virtual Laboratory ....................................... 29
3.6 Remote Virtual Laboratory ......................................... 30
3.7 Virtual Laboratory software ........................................ 31
3.8 Virtual Laboratory collaboration diagram ......................... 33
3.9 Virtual Laboratory deployment diagram .......................... 34
3.10 Virtual Laboratory use case diagram ............................. 34
3.11 Virtual Laboratory sequence diagram ............................ 36
3.12 Virtual Laboratory statechart diagram .......................... 37
3.13 Comparison between time-based and event-based control schemes . 41
3.14 Simple Petri Net .................................................... 43
3.15 General event-based architecture ................................ 45
3.16 Comparison between real and simulated signals ................ 51
3.17 Comparison between real and simulated signals ................ 52

4.1 Development methodology phases ................................. 54
4.2 Experiment design and functionality analysis ..................... 56
4.3 First modeling stage of the identification phase ................ 57
4.4 Second modeling stage of the identification phase ............... 58
4.5 Third modeling stage of the identification phase ................. 58
4.6 UML statechart-Petri Net conversion ............................. 60
4.7 Petri Net generated from the VL statechart diagram ............. 62
4.8 Coverability tree .................................................... 64
4.9 Composition, analysis, and control stages ....................... 66
4.10 Transition synchronous composition ............................... 67
4.11 Parallel place fusion ............................................... 68
4.12 Supervisory control for DES ...................................... 70
4.13 Inhibitor arc behavior ............................................. 72
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.14</td>
<td>VL development methodology</td>
<td>76</td>
</tr>
<tr>
<td>5.1</td>
<td>Development methodology diagrams</td>
<td>80</td>
</tr>
<tr>
<td>5.2</td>
<td>Robotic navigation experiment</td>
<td>82</td>
</tr>
<tr>
<td>5.3</td>
<td>C-Space Creation hierarchical chart</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>C-Space Creation class diagram</td>
<td>86</td>
</tr>
<tr>
<td>5.5</td>
<td>C-Space Creation collaboration diagram</td>
<td>88</td>
</tr>
<tr>
<td>5.6</td>
<td>C-Space Creation sequence diagram</td>
<td>89</td>
</tr>
<tr>
<td>5.7</td>
<td>C-Space Creation statechart diagram</td>
<td>90</td>
</tr>
<tr>
<td>5.8</td>
<td>C-Space Creation Petri Net</td>
<td>91</td>
</tr>
<tr>
<td>5.9</td>
<td>C-Space Creation coverability tree</td>
<td>92</td>
</tr>
<tr>
<td>5.10</td>
<td>Composed Petri Net without control</td>
<td>95</td>
</tr>
<tr>
<td>5.11</td>
<td>Coverability tree of the Petri Net without control</td>
<td>97</td>
</tr>
<tr>
<td>5.12</td>
<td>Segment of the composed Petri Net with the first controller net</td>
<td>99</td>
</tr>
<tr>
<td>5.13</td>
<td>Segment of the composed Petri Net with the second controller net</td>
<td>100</td>
</tr>
<tr>
<td>5.14</td>
<td>Segment of the composed Petri Net with the third controller net</td>
<td>101</td>
</tr>
<tr>
<td>5.15</td>
<td>Segment of the composed Petri Net with the fourth controller net</td>
<td>101</td>
</tr>
<tr>
<td>5.16</td>
<td>Petri Net modeling the behavior of the VL system</td>
<td>102</td>
</tr>
<tr>
<td>5.17</td>
<td>Coverability tree of the composed Petri Net with controller nets</td>
<td>104</td>
</tr>
<tr>
<td>6.1</td>
<td>C-Space transformation into the VL for bilateral teleoperation</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>Teleoperation system block diagram</td>
<td>110</td>
</tr>
<tr>
<td>6.3</td>
<td>Flow chart of the teleoperation system</td>
<td>114</td>
</tr>
<tr>
<td>6.4</td>
<td>Potential Field Creation hierarchical chart</td>
<td>117</td>
</tr>
<tr>
<td>6.5</td>
<td>Potential Field Creation class diagram</td>
<td>118</td>
</tr>
<tr>
<td>6.6</td>
<td>Potential Field Creation collaboration diagram</td>
<td>118</td>
</tr>
<tr>
<td>6.7</td>
<td>Potential Field Creation sequence diagram</td>
<td>119</td>
</tr>
<tr>
<td>6.8</td>
<td>Potential Field Creation statechart diagram</td>
<td>120</td>
</tr>
<tr>
<td>6.9</td>
<td>Potential Field Creation Petri Net</td>
<td>121</td>
</tr>
<tr>
<td>6.10</td>
<td>Potential Field Creation coverability tree</td>
<td>121</td>
</tr>
<tr>
<td>6.11</td>
<td>Composed Petri Net merging VL subsystems</td>
<td>123</td>
</tr>
<tr>
<td>6.12</td>
<td>Coverability tree of the composed Petri Net for bilateral teleoperation</td>
<td>126</td>
</tr>
<tr>
<td>6.13</td>
<td>Composed and controlled Petri Net for bilateral teleoperation</td>
<td>128</td>
</tr>
<tr>
<td>6.14</td>
<td>Coverability tree of the composed Petri Net for bilateral teleoperation</td>
<td>130</td>
</tr>
<tr>
<td>7.1</td>
<td>Virtual Laboratory sites</td>
<td>137</td>
</tr>
<tr>
<td>7.2</td>
<td>Virtual Laboratory configuration</td>
<td>138</td>
</tr>
<tr>
<td>7.3</td>
<td>Virtual Laboratory Guest interface</td>
<td>139</td>
</tr>
<tr>
<td>7.4</td>
<td>Planned and sensed paths</td>
<td>140</td>
</tr>
<tr>
<td>7.5</td>
<td>Time delays measured from the Host</td>
<td>141</td>
</tr>
<tr>
<td>7.6</td>
<td>Configuration of the VL for bilateral teleoperation</td>
<td>142</td>
</tr>
<tr>
<td>7.7</td>
<td>User interfaces of the VL for bilateral teleoperation</td>
<td>143</td>
</tr>
<tr>
<td>7.8</td>
<td>The robot trajectory during a teleoperation experiment</td>
<td>144</td>
</tr>
<tr>
<td>7.9</td>
<td>System behavior in a teleoperation experiment</td>
<td>145</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

A.1 Input Processing class diagram ........................................ 152
A.2 Input Processing collaboration diagram ................................. 152
A.3 Simulation class diagram .................................................. 153
A.4 Simulation collaboration diagram ........................................ 153
A.5 Sensory Conversion class diagram ....................................... 154
A.6 Sensory Conversion collaboration diagram ............................. 154
A.7 Settings/Tasks Database class diagram ................................. 155
A.8 Settings/Tasks Database collaboration diagram ...................... 156
A.9 Sensed Data Processing class diagram ................................. 156
A.10 Sensed Data Processing collaboration diagram ..................... 157
A.11 Experiment Control class diagram ..................................... 157
A.12 Experiment Control class diagram ..................................... 158
A.13 Model Library class diagram ............................................ 158
A.14 Model Library collaboration diagram .................................. 159
A.15 Experiment Database class diagram ................................... 159
A.16 Experiment Database collaboration diagram ....................... 160
A.17 Communication class diagram ......................................... 160
A.18 Communication collaboration diagram ............................... 161

B.1 Three DOF robot configuration .......................................... 164
B.2 The C-Space numeric grid ................................................ 167
B.3 Gradient descent algorithm .............................................. 168
B.4 Spline interpolation with accumulative distance .................... 172
B.5 Binarized workspace image ............................................. 173
B.6 Segmented workspace image ............................................ 173
B.7 Segmentation mask ........................................................ 174
B.8 First Hu invariant moment ............................................... 176
B.9 Trigonometrical method used to perform the path following .... 177

C.1 Differential drive control ............................................... 180
C.2 Right motor measured speed ............................................. 184
C.3 Left motor measured speed ................................................ 185
C.4 Numeric potential field .................................................. 186

D.1 Path Planning hierarchical chart ...................................... 191
D.2 Path Planning class diagram ............................................ 191
D.3 Path Planning collaboration diagram .................................. 192
D.4 Path Planning sequence diagram ....................................... 193
D.5 Path Planning statechart diagram ..................................... 193
D.6 Path Planning Petri Net .................................................. 194
D.7 Path Planning coverability tree ......................................... 195
D.8 Path Following hierarchical chart ...................................... 197
D.9 Path Following class diagram .......................................... 197
D.10 Path Following collaboration diagram ................................ 198
D.11 Path Following sequence diagram .................................... 198
D.12 Path Following statechart diagram .............................. 199
D.13 Path Following Petri Net ........................................ 200
D.14 Path Following coverability tree ................................ 201
D.15 Robot Teleoperation hierarchical chart ......................... 202
D.16 Robot Teleoperation class diagram ............................. 203
D.17 Robot Teleoperation collaboration diagram ................... 204
D.18 Robot Teleoperation sequence diagram ......................... 204
D.19 Robot Teleoperation statechart diagram ....................... 206
D.20 Robot Teleoperation subsystem Petri Net ...................... 207
D.21 Robot Teleoperation coverability tree ......................... 208

E.1 Complementary observer system .................................. 210
E.2 Observer user interface ............................................. 210

F.1 Final Virtual Laboratory layout .................................. 211
F.2 Proximity and direction of obstacles ............................. 213
F.3 Virtual obstacle insertion ......................................... 214
F.4 Virtual obstacle warning .......................................... 215
F.5 Virtual obstacle avoiding ......................................... 216
F.6 Robot teleoperation control ...................................... 217

G.1 Decreasing the number of transmitted images ................. 220
G.2 Decreasing transmitted images in the VL for mobile robotics 222
G.3 Decreasing transmitted images in the VL for bilateral teleoperation 223
List of Tables

2.1 Comparison of proposed approaches in existing works .................................. 20

5.1 C-Space Creation functionality specifications .............................................. 84
5.2 Joined place numeric labeling and its description ......................................... 94
5.3 Numeric transition labeling and its description ............................................ 96
5.4 Places added by controller nets ............................................................... 103

6.1 Definition of variables used in the block diagram ......................................... 110
6.2 Equations for joystick position conversion into desired velocity ................. 111
6.3 Equations for proximity conversion into potential field velocity .................. 112
6.4 Potential Field subsystem specifications .................................................. 117
6.5 Place numeric labeling and its description ................................................. 124
6.6 Numeric transition labeling and its description ........................................... 125
6.7 Places added by controller nets in VL for teleoperation ................................. 129
6.8 Transitions added by controller nets in VL for teleoperation ......................... 129

7.1 General hardware description and specifications ........................................... 135
7.2 Path analysis results .................................................................................. 140
7.3 Time analysis results .................................................................................. 141

C.1 Dynamic behavior of the Khepera minirobot ............................................... 183

D.1 Path Planning functionality specifications .................................................... 190
D.2 Path Following functionality specifications ................................................... 196
D.3 Robot Teleoperation subsystem specifications .............................................. 205
Chapter 1

Introduction

The last decade was characterized by a strong and sustained progress in information and communication technologies. This progress has provoked a profound impact on education and research in different fields.

In education, the development of information and communication technologies has created remote and global distributed resources of instruction and information, modifying traditional ways of teaching and learning. As a consequence, instruction and information are no longer restricted to classrooms and libraries, and new learning paradigms have appeared. Educational institutions are integrating the new distance-learning paradigm into their ways of teaching.

Unfortunately, distance learning lacks practical activities in the laboratory. Students that are using this new paradigm do not have the same opportunities to develop their practical skills, like those that have access to these facilities [Licks et al. 2000]. These students could have training deficiencies that would decrease their possibilities of being immediately incorporated into the work force. Laboratory exercises are part of any university engineering education to help students get acquainted with industrial equipment, measurement tools, and procedures [Karady et al. 2000]. In distance education, this important part has to be offered in special courses, where students from different places have to meet in real laboratories.

On the other hand, these information and communication technologies were used to communicate and share information, closing the gap between geographically distant research groups. However, building and mounting laboratory facilities with sophisticated and expensive equipment still remains a problem to be solved because of high budget requirements. Moreover, cost increases when personnel transportation is provided [de Queiroz et al. 1998]. Consequently, these laboratories are scarce. Therefore, it is essential that researchers, not physically present at the laboratory facility, use the laboratory equipment.

The Virtual Laboratory (VL) concept emerges to offer wide access over the
Internet to these laboratories in a way that help people develop their practical skills to carry out experiments or perform research using sophisticated equipment. VL applications allow the users to mount, define and perform experiments, collect experimental data, and analyze the results as if they were physically present in the laboratory.

Widely spread availability and the relative low cost of Internet have motivated recent growth of VL implementations. This growth has provided on-line Internet resources to perform experiments in several engineering areas: control [Irawan et al. 2001], bioinformatics [Alonso et al. 2001], chemical engineering [Moros et al. 2002], electronics [Cheng et al. 2002], manufacturing [Ahn et al. 2002], signal processing [Dow et al. 2003], robotics [Cosma et al. 2003], electrical engineering [Basher et al. 2004], physics [Kawabata et al. 2004], instrumentation [Pisani et al. 2004], electromagnetics [Magistris 2005], and many others.

Considerable research placed on the robotic teleoperation area, where specialized and expensive equipments are simulated or controlled remotely via the Internet, concentrates on the proposal of new architectures for remote teleoperation [de Queiroz et al. 1998, Dalton and Taylor 2000, Goldberg et al. 2000, Saucy and Mondada 2000, Brady and Tarn 2000, Brady and Tarn 2001, Lim et al. 2003], and remote simulation [Hirukawa et al. 1997, Schulz et al. 1999, Munoz-Gomez et al. 2003]. Nonetheless, these works did not propose any way of methodically building and developing new teleoperation implementations that fill VL functionalities needed to perform true experiments [Hicks 1982].

Several works have modeled these teleoperation systems as VLs. These efforts have created development frameworks taking into account VL service requirements [Guimaraes et al. 2003a, Guimaraes et al. 2003b], or procedures for teleoperation tasks as software components [Bottazzi et al. 2002, Amoretti et al. 2003]. However, these development frameworks generate a static model, which is specific to the application and does not model their dynamic behavior. These frameworks are based on CORBA-complaint (Common Object Request Broker Architecture) software components and use the CORBA’s event service management to provide synchronization between the components. Thus, dynamic behavior is hidden and ignored under the CORBA implementation.

Other research works dynamically modeled and analyzed Internet-based bilateral teleoperation systems [Xi 1993, Xi and Tarn 1999, Elhajj et al. 2001, Elhajj et al. 2003, Elhajj et al. 2004]. These works allowed the development of telerobotic applications that provide real-time sensory information (video, force, temperature) to the user over the Internet, using an event-based control approach. Nevertheless, this modeling approach does not consider software engineering design; they take no heed of the control and coordination of several subsystems that are necessary for multiple functionalities of a VL application.
1.1. Problem Statement

To solve the need for a complete modeling and systematic development to design and implement VLs in the teleoperation area, this thesis proposes a general and modular model. This model is described using a standard software engineering representation creating a general framework, which provides a skeleton customized by the developer, based on experiment specifications. To systematize the VL design, a development methodology that takes the general framework as starting point, is also proposed. This methodology allows the developer to customize the framework, transform it into a dynamic modeling formalism, perform analysis and control of the experiment behavior, and generate a complete VL design.

1.2. Thesis Statement

This thesis presents a general and modular VL model for telerobotics over the Internet and a methodology that uses this model as reference to develop, analyze, and design a VL application. The goal is to create a generic and formal framework to lead the appropriate configuration of a specific VL application and to transform this framework into another dynamic modeling formalism, providing the necessary tools to perform a complete analysis and control design of VLs.

The research work focuses on proving that:

The static and dynamic design of a VL for the teleoperation area, its elements, components, and its interrelationships can be specified together, using the proposed model as a basis to create a reference framework that can be customized, based on the experiment specifications.

A VL for telerobotics can be designed as a no-time referenced system to maintain its stability and synchronicity over the Internet. An event-based control architecture is proposed to fulfill these requirements and to provide better interaction between the
user and the experiment.

The multiple functionalities of a VL for experimental teleoperation are performed by several subsystems in the application. These subsystems must be sequentially performed to achieve the experiment goals. Therefore, it is important to apply a method of control design on these subsystems when they are integrated into a VL. The design, analysis, and control of these functionalities can be made by performing a modular development based on the repetitive instantiation of the model and merging the subsystems as non-deterministic Discrete Event Systems (DES). Furthermore, this modular development allows reusing software components to create new VL applications or enhance their capabilities.

The development of VLs for telerobotics must be methodically performed to allow modeling, analyzing, and designing of these kinds of applications. The proposed methodology uses the model as a reference framework to model, analyze, and design VLs. This development methodology performs the structural and behavioral design using two formal representations: the Unified Modeling Language (UML) for structural definition and Petri Nets for behavioral analysis and control design.

VL applications can be successfully implemented using the design generated by the proposed methodology. Formal notations used by the methodology are compatible with the object-oriented programming paradigm. Therefore, the design can be directly implemented by developers using object-oriented languages and standard interconnection technologies. In this work, two VLs are designed, implemented, and tested to provide experimental results to prove the proposed hypothesis.

1.3 Methodology

This dissertation proposes a general and modular VL model that represents the abstract description of processes and devices that could be enclosed in a VL system. The model is specifically oriented to design VLs for robotic teleoperation and can be fully or partially implemented depending on experiment specifications. The modular characteristic of the model allows reusing previously designed components, to increase the capabilities and functionalities of the system, or to design new experiments.

The abstract model is composed of three elements: Guest, Media, and Host. The Guest is a computer-based system that shows user interfaces allowing the user to interact with the experiment. The Host system executes tasks requested by the user accessing simulated or physical equipment. The Media establishes the communication link between Guest and Host via computer digital networks and data transport protocols.

Guest represents the necessary software and hardware that allow the user to
introduce commands and provide sensory feedback regarding the experiment. *Guest* software is contained in a computer that has connected input devices, such as keyboards and joysticks, and output devices, such as monitors and transducers. The user interacts with the input devices to introduce commands, which are processed and transmitted by an input processing procedure to the *Host* system through communication procedures of *Media*. Sensory feedback regarding the experiment is provided by output devices. These devices receive the appropriate signal from a sensory conversion procedure that receives this sensory information from *Host* via *Media*. *Guest* includes a simulation procedure that represents processes to recreate the experiment using simulated models of laboratory equipment and provides an immediate or predicted sensory feedback. *Guest* also has a database procedure to retrieve experiment settings or previously performed tasks.

Meanwhile, *Host* models software and hardware to perform tasks required by the user. The *Host* software is enclosed in a computer that has connected sensor devices, such as cameras and proximeters, and mechanisms and control devices, such as manipulators and mobile robots. The *Host* software is represented by several procedures that allow remote control and perception of experiments. The control procedure processes user commands received via *Media* and sends its output action to a command conversion procedure that generates appropriate commands for mechanisms and control devices. This control procedure can work in an open-loop control fashion using user commands as its reference or in a closed-loop control when it also receives processed data from a sensed data processing procedure that acquires sensory information from sensor devices. The sensed data processing procedure also provides sensory data, through *Media*, to the *Guest* procedures. *Host* includes a software library that contains simulated models used by the *Guest* simulation procedure. A database is included in *Host* to represent procedures to store and retrieve acquired experimental data for further analysis.

The abstract model is translated into UML (Unified Modeling Language) formalism to describe and define in detail its elements and components and relationships between them. UML is an object-oriented language capturing the idea behind complex systems such as VLs; it is a standard language for the software engineering design. UML gives several points of view and captures the static structure, dynamic behavior, and functionalities by means of graphic elements and rules combined into diagrams, which are known as a model. This conversion defines an object-oriented framework, which is used as a generic structure that can be adapted to experiment specifications.

Since UML does not have quantitative and qualitative analytical tools to capture the dynamic behavior of VL systems, the model is converted into Petri Net formalism. Petri Nets are graphical and mathematical modeling tools with attractive features based on their ability to describe and study systems that are characterized as: concurrent, asynchronous, distributed, parallel, non-deterministic, or stochastic. Furthermore, Petri Nets have the ability to set up state equations, algebraic equations,
and other mathematical models governing the behavior of the system. Therefore, the methodology allows the developer to perform dynamic modeling, analysis, and control of the experiment components to structurally and behaviorally design a VL application.

A teleoperation architecture is also introduced and described. This architecture combines VL model elements with the event-based control and planning approach [Elhajj 2002, Xi 1993] to generate control requirements and procedures for stability and validation of synchronization in VL designs.

Figure 1.1: Conceptual development of the general and modular model for VLs for telerobotics over the Internet.

To systematize VL design, a development methodology is also proposed, taking the generic framework as the basic support. The methodology starts performing an experimental design and analysis, which take into account the objectives of a true experiment [Hicks 1982] and the procedures that will be used to achieve these objectives. The experiment process flow chart and its complete description are analyzed to identify main functionalities to mount, define, and execute the experiment. This analysis generates several tables with processes and devices, which include their specifications and functions. These tables are summarized in hierarchical charts for the three main functionalities with the processes and devices and their corresponding functions.
1.3. Methodology

Experiment Design and Functionality Analysis

Experiment Subsystems Identification in the VL Framework

Conversion into a Dynamic Model and Analysis

Composition, Analysis, and Control of the Dynamic Model

VL Complete Design

Figure 1.2: Development methodology to structurally and dynamically model, analyze, and design VLs for telerobotics.

Modeling the experiment, using the VL generic framework as a guideline, is the second phase of the methodology. Here, main functionalities are considered as subsystems, which are related to a complete instance of the generic framework to allow modular development. Processes and devices summarized in the generated tables and charts are identified using the object-oriented paradigm. This identification process takes as reference, class definitions, object associations, and operation methods described in the generic framework. This phase models processes and devices as objects, their specific functions as operation methods, and their specifications as class attributes. Then, a customized VL framework for each subsystem is built.

Following the methodology, the third phase focuses on the conversion of customized VL frameworks into Petri Net formalism to perform the dynamic analysis for every subsystem. UML to Petri Net conversion is based on the foundations of state machines of UML statechart diagrams. Once the VL subsystems are converted, a behavioral analysis can be performed. This analysis checks several behavioral properties in Petri Nets representing VL subsystems. Properties being checked are related to synchronization, deadlocks, and signal accumulation in VL subsystems. If behavioral
problems exist, then the dynamic design has to be reviewed.

Once the VL subsystems are verified, the final step merges them to integrate a complete VL system and performs an analysis on the composed system; if the system does not behave correctly, then design methods to synthesize control structures are used. This merge is based on the synchronous composition [Kumar and Holloway 1996], and parallel place fusion reduction rules [Murata 1989] of Petri Net formalism. Since this composition does not guarantee that subsystems will be executed in an adequate way, the composed VL system is analyzed to verify that VL subsystems perform their functions correctly. If these subsystems are not executed in a sequential and synchronized way in the VL dynamic design, then supervisory control design methods for Discrete Event Systems (DES) are used on the VL dynamic design. A method to synthesize controller nets using inhibitor arcs for safe ordinary Petri Nets is proposed and introduced. The proposed method enforces constraints of the logical form if-then. Finally, the complete Petri Net design is obtained.

When the methodology ends, the VL is described statically and dynamically by means of several diagrams using UML and Petri Net formalisms. The customized UML framework specifies software functionality and leads its implementation on programming languages that are based on the object-oriented paradigm. Petri Net modeling and its analysis provide a compatible design that takes into account the dynamic behavior of the VL system. Therefore, the diagrams generated by the methodology can be directly used by the developer.

The VL model and its development methodology were verified by the experimental results of two implemented systems. The first system is based on a didactic experiment for mobile robotics, which is transformed into a VL using the model-based development methodology. The results of several experiments using an Internet connection between two geographically distant sites showed stability, accuracy, and performance of the implemented system. The second developed system is a new bilateral teleoperation experiment that implements an event-based control for robotic navigation providing visual and haptic feedback. This experiment takes advantage of the modularity of the framework by using several components generated for the previous VL and by adding new sensory feedback capacities. Experimental results were acquired using an Internet2 connection to show stability and performance of the teleoperation system.

1.4 Scope of the Thesis

The research performed in this thesis concentrates on providing a model-based general framework to guide the design and development of VLs for the specific area of Internet-based telerobotics.

The main theoretical contributions of this research include: an abstract VL
model to define a general framework using a software engineering notation and a novel methodology to customize the framework, depending on experiment functionalities and specifications. This methodology transforms the customized framework into another modeling formalism to allow the developer to perform behavioral analysis, which validates the system stability before its implementation. These contributions also include an event-based teleoperation architecture, definition of event-based simulation design requirements, and a method to control DES (Discrete Event Systems), among others.

The experimental results of the remote interaction performed with two developed VL applications for telerobotics, which were designed using the proposed methodology and the reference framework, verify the proposed theory. These applications are: a VL for mobile robotics and a VL for bilateral teleoperation. The first implementation is created as a didactic and research test bed for robotic navigation. The second one is designed by reusing software components from the first implementation. This VL is created as a research test bed for experimentation on enhanced sensory feedback (haptic and visual) in bilateral teleoperation. To the best of our knowledge, the second implementation is an one-of-a-kind system that has not been developed previously. Furthermore, a complementary system is also developed using a mobile camera to provide an enhanced visual feedback of the experiments.

The experimental results of both VL implementations were analyzed to provide concluding remarks regarding their stability, accuracy, and performance over the Internet. These results were obtained using geographically separated test beds located approximately 1,050 km away and Internet and Internet2 links with multiple jumps and different delays. Performance analysis done of successful robotic navigation and bilateral teleoperation experiments confirmed all of the theory developed and illustrate its benefits.

1.5 Dissertation Overview

This document presents the detailed description of the general and modular VL model. It shows the model translation in two modeling formalisms to support the development of VLs for telerobotics. It also describes two VL developments using the proposed theory, besides analyzing experimental results to demonstrate theory benefits and provide conclusions of research work done. Chapter 2 overviews the existing approaches that are used by the VL model to solve Internet challenges. It describes the difficulties faced when the Internet is used as a communication medium for telerobotics, summarizes the different development approaches taken in previous works, and describes their limitations. Chapter 3 presents the proposed model and outlines its composition as a general UML framework. It presents a detailed explanation regarding the VL concept and the proposed abstract model, describes its conversion into the UML and Petri Net formalism, and proposes a teleoperation architecture based on model components. Chapter
4 presents the development methodology for modeling, analyzing, and designing VL applications. It explains in detail the four phases that compose the methodology: Experiment Design and Functionality Analysis; Experiment Subsystem Identification in the Reference Framework; Conversion into a Dynamic Model and its Analysis; Composition, Analysis, and Control of the Dynamic Model. Chapters 5 and 6 describe two different system developments using the general framework and the proposed methodology to model, analyze, and design two telerobotic experiments as VL applications. The first development takes an experiment for mobile robotics and develops it as a VL. The second development creates and designs a new experimental VL application, which takes advantage of the modular components of the first implementation. The technical development and experimental results of both VL implementations are presented in Chapter 7. Finally, thesis conclusions, contributions, and future directions are presented in Chapter 8.
Chapter 2

Overview of Previous Work

The field of Internet-based telerobotics embodies mechanics, electronics, computer, control, and network communication technologies. Existing applications in this field share several similar challenges; the greatest challenge is time delay. Early implementations were built based on ad hoc architectures for time-delayed teleoperation [Taylor and Travelyan 1995]. Meanwhile, other applications for bilateral teleoperation were based on passive circuit models, virtual time methods, delay prediction, and recently on nontime-referenced systems [Kosuge and Murayama 1997, Liu et al. 2002, Elhajj et al. 2000a]. These approaches focused on reducing the impact of time delay on stability and efficiency of teleoperation systems.

Another challenge in the telerobotic field is the creation of distributed robotic resources over the Internet. Several implementations were developed, based on standard general-purpose interconnection technologies called middleware [Hori et al. 1999, Dalton and Taylor 2000]. These applications integrate heterogeneous systems and reduce the programming efforts hiding, from developers' view, details regarding communication among the distributed resources. This approach leads to the creation of collaborative robotic systems over the Internet.

On the other hand, the VL concept gained significance and caused multiple research works to introduce different development approaches. Early VL implementations were built to remotely operate measurement equipment residing in a laboratory facility that allowed experiments to be performed with them [Daponte et al. 1994, Ko et al. 2000]. Later on, a number of telerobotic applications were implemented as VLs [Munoz-Gomez et al. 2003, Guimaraes et al. 2003a]. These VLs were arranged to allow remote experimentation, collection of information, and validation of solution methods for research or didactic telerobotics purposes. Some of these laboratories used the object-oriented programming paradigm to build software development frameworks based on middleware technology.

The works mentioned in this chapter are the basis for building the proposed model.
Chapter 2. Overview of Previous Work

and its systematic method. The model joins tested Internet-based teleoperation strategies with generic components; it is described using the object-oriented paradigm to produce modular designs.

The following sections describe the difficulties found when the Internet is used as a communication medium for teleoperation, and provide a summary of the existing approaches to develop telerobotic implementations over the Internet, including those being built as VLs. These approaches are divided into four groups: time-delayed teleoperation, bilateral teleoperation, distributed telerobotics, and VLs. Finally, the limitations of these development approaches are described in detail.

2.1 The Internet as a Communication Medium

Several difficulties are present when the Internet is considered as a communication medium for robotic teleoperation. The main effects of these difficulties are instability and lack of synchronization on telerobotic systems with closed-loop control, such as bilateral teleoperation systems for real-time remote control.

The Internet can provide a reliable communication link when protocols, such as Transport Control Protocol (TCP) [Comer 2000, Stevens 1994], are used. Some Internet protocols provide an ordered flow of data packets and do not lose them (as long as no disconnect occurs). However, the Internet does not provide a guarantee of Quality of Service (QoS), since a disconnect may occur at any time, and communication parameters, such as bandwidth and delay can fluctuate in a significant way because of non-deterministic behavior of network traffic [Leland et al. 1994, Beran et al. 1995, Paxson 1999, Gao and Rubin 2002].

This lack of guarantee results from Internet topology, which is implemented using heterogeneous hardware and communication protocols. Thus, data packets traveling between two hosts can go through diverse routes with different communication qualities. Infrastructure failures, time-of-day patterns, temporary outages, fluttering, erroneous routing, and routing loops are some of the existing routing pathologies that exist on the Internet [Paxson 1997].

Therefore, the inter-arrival time of packets cannot be accurately estimated and it generates an unpredictable latency in the net. Disconnects at any time and indefinite random delay are the result of these problems. Several studies have shown that the delay cannot be modeled with acceptable accuracy [Paxson and Floyd 1995, Floyd and Paxson 2001]. This delay is non-fixed and random; it is not the same in both directions and lacks an upper bound [Oboe and Fiorini 1997, Bolot et al. 1990].
2.2 Time-delayed Teleoperation

In general, implemented Internet-based teleoperation applications allow operating robotic devices in simulated or remote environments. In [Hirukawa and Hara 2000] the teleoperation systems were grouped into two types: direct and indirect. In the first type, the operator directly controls a real robot. In the second type, the operator plans robot tasks interactively through computer simulation of the robot and its environment; planned motions were sent to control a remote robot afterward. Some works applied these two approaches to reduce the impact of time delay over the Internet since the developed architectures provided remote access to real robotic devices, simulated models, or a combination of both to predict and obtain an immediate feedback of the robot status.

2.2.1 Direct Teleoperation

The first telerobotic implementations over the Internet used open-loop control architectures for direct teleoperation. Although open-loop systems are not optimal, they can tolerate great delays, while the user visually closes the control loop. However, as delay increases, the operator's efficiency drops dramatically [Ferrel 1963]. In such a situation, operators adopt "move and wait" strategies to minimize errors.

The Internet offers diverse communication technologies available to execute requests remotely in an open-loop fashion. Early web-based telerobotic applications used the HyperText Transport Protocol (HTTP) with the Common Gateway Interface (CGI) of web servers to teleoperate robotic manipulators [Goldberg et al. 1994, Goldberg et al. 1995, Taylor and Travelyan 1995, Golberg et al. 1996] and mobile robots [Simmons et al. 2000, Michel et al. 2000, Saucy and Mondada 2000]. The commands issued were introduced using input formats of HyperText Marking Language (HTML), and visual feedback was performed by refreshing the HTML page after certain time intervals.

These applications had several limitations, such as: user interfaces with poor interaction capabilities, static visual feedback, and slow response. These limitations were overcome by the use of the Java programming language. Java enables operating with network connections means that can be implemented using TCP and User Datagram Protocol (UDP), among others. The ability of Java applets to run on Virtual Machines (VM) of Internet browsers was used to access compiled libraries from remote sites and generate sophisticated Graphical User Interfaces (GUI) without installing any software application at the client site [Backes et al. 1998, Hirukawa and Hara 2000].

When the operator uses open-loop architectures and remotely performs a task, long communication delays prevent detection of problems since there is not enough time to correct them. To overcome this difficulty, the teleoperation system is provided with au-
tonomy to detect and solve problems immediately. The idea is to process information from sensors in the computer system that control the robot to make an internal closed-loop control. This sensor’s information allows detecting, correcting, and avoiding errors to improve the system’s performance. This approach was developed for general teleoperation systems as a supervisory control [Sheridan 1987, Sheridan 1992, Sheridan 1993], and produces system autonomy under human supervision [Hayward 1988]. In this approach, the system connected to sensors and robotic devices can execute low level tasks based on high level commands issued by the operator. The human operator supervises the task execution to intervene when the task is not respecting the original plan, assuming control when necessary [Grange et al. 2000]. This approach gives autonomy to the teleoperation system; therefore, a trade off between the system automation and user interaction is needed to perform a true experiment [Hicks 1982].

2.2.2 Indirect Teleoperation

The way to improve user performance is to provide an immediate sensory feedback. Because there is communication delay, this feedback cannot come from the remote site where the robotic device is located. Instead, it has to be generated by the operator’s computer. The idea is to insulate the operator from communication delay by providing a simulated feedback.

Three architecture prototypes to simulate robotic devices were proposed by [Hirukawa et al. 1997]. These prototypes were developed, based on Virtual Reality Modeling Language (VRML) technology. VRML technology is capable of accessing software libraries containing simulated models from remote sites via the Internet. These models can be downloaded by plug-in extensions of web browsers and visualized in a 3-D fashion by the operator’s computer [Michel et al. 2000]. Furthermore, they can contain interactive operations to control its behavior in a simulated environment and obtain other sensory information, such as force reflection.

On the other hand, task simulation can reduce failure when it is used to predict robotic behavior before performing its task. The user can perform a simulated test and observe immediate reactions on the simulated device, taking the correct actions to perform the successful task using the real robotic device (indirect teleoperation). A teleoperation system controlling a robot manipulator was implemented with two subsystems for indirect and direct teleoperation in [Yang et al. 2004].

An indirect teleoperation was proposed by the teleprogramming paradigm [Funda et al. 1992, Paul et al. 1993], based on supervisory control. This paradigm describes symbolic commands, which are introduced in a code recognized by the system to allow programming robotic devices online. Each command, introduced by the user acting now as a programmer, is processed immediately and executed in the experiment simulation. Then, the corrected code is sent to the remote computer. The idea is to send a minimal code by debugging the code, using simulation and programming the
2.3. Bilateral Teleoperation

Force and visual information are the most common sensory feedbacks for improving teleoperated systems. Providing force reflection feedback can couple the operator kinesthetically to the remote environment and increase the sense of telepresence. Teleoperation systems for bilateral control, using closed-loop architectures, were developed to provide this sensory information. Nevertheless, delays on the order of a tenth of a second can destabilize these bilateral teleoperation systems [Anderson and Spong 1989a, Anderson and Spong 1989b].

The impact of time delay on bilateral teleoperation over the Internet has increased, due to its non-deterministic behavior [Leland et al. 1994, Beran et al. 1995, Paxson 1999, Gao and Rubin 2002]. A research work based on modeling a passive circuit was developed by [Kosuge and Murayama 1997] to assure system stability. Later, a bilateral teleoperation architecture, using a virtual time delay method, was proposed by [Kikuchi et al. 1998]. In this application, control information was sent with time stamps to keep time delay uniform. More recently, a delay prediction approach was proposed by [Mifakhrai and Payadeh 2002]. This approach used an autoregressive model to forecast future delay values. Another approach to predict delay boundaries was presented by [Liu et al. 2002]. These approaches have limitations because some of them consider fixed delays with upper bounds or require previous knowledge of delay behavior. Under such considerations they cannot usually be accepted for some telerobotic applications that require high reliability and safety.

A method for action synchronization and control of telerobotics over the Internet was proposed by [Xi and Tarn 1999]. This method deals with random time-delay in the Internet using a sensor-based referenced action control scheme. Another application for bilateral teleoperation, based on the same control scheme, was presented in [Elhajj et al. 2000b]. This research introduced a real-time event-based control with haptic feedback to teleoperate a mobile robot over the Internet. In [Elhajj et al. 2000a, Elhajj et al. 2001] an event-based control for cooperative teleoperation was designed using a Petri Net model. Subsequently, other applications for cooperative control, multi-robot formations, tele-coordination, and supermedia
(video, force, temperature) enhancement were developed, based on the same approach [Lo et al. 2003, Elhajj et al. 2002b, Elhajj et al. 2002a, Elhajj et al. 2003, Elhajj et al. 2004]. Until now, event-based systems have proven their stability and potential to feed the user with rich sensory information. However, the performance of these systems is still affected by longer delays.

### 2.4 Distributed Telerobotics

Another approach to build Internet-based telerobotics applications considers the connected robotic devices as distributed objects. In [Hori et al. 1999], a distributed object computing approach was introduced for Internet-based telerobotics. This research work evaluated a simulated telerobotic application based on the Common Request Broker Architecture (CORBA). It used the object-oriented paradigm of computer programming to implement interoperability among networked robots via the Internet InterOrb Protocol (IIOP). Its ability to interconnect clients to other servers and clients, known as peers, results in distributed systems that allow interaction and collaboration among peers.

A comparison of middleware technologies to create distributed robotic systems, such as CORBA, Remote Method Invocation (RMI), and Message-Oriented Middleware (MOM) was provided in [Dalton and Taylor 2000]. These middleware technologies hide particular aspects of the distributed objects (heterogeneous robotic systems) and present only essential aspects to the software developer.

The CORBA-based distributed telerobotics was used to create a development framework introduced by [Bottazzi et al. 2002]. This research described a software framework that exploits advanced CORBA features, such as the Asynchronous Method Invocation (AMI) and its real-time priority feature to support concurrent actions in the telerobotic system. It partially used the Unified Modeling Language (UML) to model collaboration between objects. Subsequently, in [Amoretti et al. 2003], three implementations of a sensory data distribution system were evaluated to validate the CORBA-based software framework. In this research, UML class diagrams were used to model the objects implicated in a peg-in-hole task. However, these works did not provide a way to validate the dynamic behavior of their final design. They encouraged the development of VL applications that totally depend on new features in middleware interconnection technology or that completely trust on the multi-session management of this technology.

On the other hand, in [Hou and Su 2004], a mission/behavior-based robotic teleoperation scheme was developed to produce a distributed perception strategy. In the implementation of this teleoperation scheme, the Serialized Object method from Java was used to guarantee the software object transfer over the Internet.
2.5 Virtual Laboratories

Many innovative development approaches were proposed to build Internet-based telerobotic systems, with its potential applications in areas such as distance education and research. In this context, Internet-based telerobotic systems evolved into VLs for didactic and research experimentation to access sophisticated and expensive robotic equipment.

Multiple implementations in the Internet-based telerobotic area were built, based on time-delayed teleoperation architectures. In [de Queiroz et al. 1998], a client-server architecture was used to implement a VL for robotic and computer vision. Another client-server approach was proposed by [Gordillo 1998] to design a VL for robotics and manufacturing. A web-based architecture was proposed in [Licks et al. 2000] for distance education. Moreover, a multilevel abstract architecture was proposed in [You et al. 2001] to divide complex telerobotic systems into seven task levels; from a low-level sensory measurement to high-level task planning and enhanced sensory feedback. However, these approaches are specific to their purpose and cannot be applied generically.

Several works introduced development approaches to implement VLs. In [Daponte et al. 1994], the object-oriented approach was introduced in conjunction with a real-time architecture to implement measurement instruments in a VL environment. A network of interoperable manufacturing software agents, stratified into three layers for prototype designing, planning, and fabricating was developed and implemented in an integrated Computer-Aided Design, Process Planning, and Manufacturing (CAD/CAPP/CAM) system [Smith and Wright 1996]. Other VLs for measurement were created for educational purposes [Ko et al. 2000, Joler and Christodoulou 2001]. In [Kassouf et al. 1999, Levert and Pierre 2000], the distributed object computing approach was used to define a conceptual model of the communication platform for VLs over the Internet. The same approach was used by [Afsarmanesh et al. 2001] to introduce a software architecture for VLs in a distributed computer environment. A resource management system, using a client-server architecture, was proposed by [Saliah et al. 2000]. A software development methodology was presented in [Scherp 2002] to generate process workflows for simulated experimentation. These works defined abstract and conceptual requirements to develop VLs, but they did not provide a systematic development method.

Recently, some object-oriented approaches were proposed: a Virtual Reality object-oriented architecture was presented to implement VLs and perform practices in simulated robotic manufacturing environments [Munoz-Gomez et al. 2003]; in [Freire et al. 2004], a VL for robotics with a multimedia environment was implemented, using the object-oriented Java 3D Application Programming Interface (API); also a development framework, based on distributed objects and relying on CORBA-compliant software components, was developed by [Guimaraes et al. 2003a,
Guimaraes et al. 2003b]. Therefore, this framework depends on the CORBA Component Model (CCM), and employs component-based software architecture to develop component frameworks for several services provided by the application.

2.6 Limitations of Existing Works

The existing development approaches in the Internet-based teleoperation area focused on solving time-delay difficulties, increasing the sense of telepresence, or generating distributed resources over the Internet. A summary and comparison among the proposed development approaches in existing works are shown in Table 2.1.

Time-delayed teleoperation implementations were developed, based on \textit{ad hoc} open-loop architectures to overpass time-delay difficulty. These teleoperation systems were designed to take advantage of specific information transfer, simulation, and VM processing technologies, such as HTTP-CGI, VRML, and Java applets. Furthermore, these research works did not propose any way to model their complete structure or their behavior, nor provide a methodology to systematically build them as VLs.

Approaches introduced in research works for bilateral teleoperation were created to provide stability to bilateral closed-loop systems from a control point of view. The passive characteristic of equivalent electric circuits was exploited to model bilateral teleoperation systems and generate control laws in discrete time domains. In contrast, event-based bilateral teleoperation systems were designed to create stable and synchronized non-time referenced systems over the Internet. They provided ways to model and develop bilateral teleoperation applications using Petri Nets as a tool for their dynamic representation and analysis. These approaches generated models specifying the dynamic behavior of closed-loop master-slave systems. Therefore, these approaches are insufficient to structurally and behaviorally model complex systems with multiple control loops such as VLs for telerobotics.

Distributed telerobotic approaches used middleware interconnection technologies to create implementations that satisfy general requirements of distributed computing, e.g., location transparency and interoperability. Although recent works using this approach proposed a software framework to provide a systematic way to develop Internet-based telerobotic applications, these frameworks depend on advanced features of specific versions of the CORBA platform to efficiently manage multiple concurrent actions to teleoperate robots and sensors from remote clients. They created appropriate software structures, but did not provide a systematic way to analyze and design its dynamic behavior.

Taking into account the above-mentioned limitations, the absence of a model that joins different approaches to provide a generic basis for VL systematic development is evident. Thus, the purpose of this thesis is to support systematic development of
2.7. Overview Conclusion

VLs for telerobotics, introducing a generic and modular model that translates into a standard modeling language defining a reference framework used to systematize the design, analysis, and implementation of VLs for telerobotics.

The proposed model is described, using a standard object-oriented modeling language. It can be customized, based on experiment specifications, and transformed into graphic and mathematical formalizations to analyze and evaluate its behavior before its implementation.

The development methodology uses an object-oriented model description as a reference to structurally and behaviorally model and analyze VLs for telerobotics, providing designs that are compatible with existing object-oriented programming platforms and middleware interconnection technologies, such as CORBA, without depending on these technologies for development.

2.7 Overview Conclusion

This chapter has reviewed existing works to design and implement Internet-based telerobotic systems. It has discussed different development approaches presented by existing works. These works created ad hoc architectures to solve specific time-delay Internet issues, or to solve interoperability problems that occur when heterogeneous robotic devices are distributed over the Internet and are teleoperated around the world.

Some of the reviewed works modeled only the structure or dynamics of Internet-based telerobotic systems. Existing research works for bilateral teleoperation introduced dynamic models based on passive circuits or event-driven systems. However, these works did not provide a way to design and model their software and hardware structure. In contrast, existing works for distributed telerobotics and VLs introduced structural models creating software frameworks based on the object-oriented paradigm and UML. However, these works were middleware-dependent; they built the behavioral design based on new features of middleware technologies, such as CORBA. Furthermore, their dynamic design was not analyzed and validated in any way. Table 2.1 summarizes essential features of these works.

Therefore, existing works did not provide a systematic approach to structurally and dynamically model and design VLs for telerobotics. Furthermore, their development approaches did not take into account VL functionalities needed to perform true experiments.
Table 2.1: Comparison of proposed development approaches in existing works.

<table>
<thead>
<tr>
<th>Previous Works</th>
<th>Ad hoc Arch.</th>
<th>Dynamic Modeling</th>
<th>Structural Modeling</th>
<th>CORBA Dependent</th>
<th>Software Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Teleoperation:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goldberg et al. 1994</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taylor and Travelyan 1995</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goldberg et al. 1996</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backes et al. 1998</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simmons et al. 2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Michel et al. 2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hirukawa and Hara 2000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indirect Teleoperation:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stein and Paul 1994</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hirukawa et al. 1997</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Friz et al. 1998</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schulz et al. 1999</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yang et al. 2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bilateral Teleoperation:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kosuge and Murayama 1997</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kikuchi et al. 1998</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elhajj et al. 2001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mifakhrai and Payadeh 2002</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Liu et al. 2002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elhajj et al. 2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lo et al. 2003</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elhajj et al. 2004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed Telerobotics:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hori et al. 1999</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dalton and Taylor 2000</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Botazzi et al. 2002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amoretti et al. 2003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hou and Su 2004</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Laboratories:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Daponte et al. 1994</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smith and Wright 1996</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>de Queiroz et al. 1998</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gordillo 1998</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kassouf et al. 1999</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Licks et al. 2000</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ko et al. 2000</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>You et al. 2001</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Afsarmanesh et al. 2001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Munoz-Gomez et al. 2003</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guimaraes et al. 2003</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Freire et al. 2004</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 3

Virtual Laboratory Modeling

This chapter proposes a generic and modular model to guide the development of VLs for Internet-based telerobotics issues. The proposed model is taken as a basis to systematically generate a complete software and hardware design of VL implementations for specific telerobotic experiments.

As already mentioned in the previous chapter, the generic and modular model is defined, using two standard and formal modeling techniques to generate a complete structural and behavioral description, and provide analytical tools needed to design VLs. The model is also related to a general teleoperation architecture to create control design procedures that assure stability and synchronization over the Internet. This architecture combines VL model components with event-based control and planning theory.

The first sections introduce the VL concept, describe model elements, provide an abstract perspective of the VL model [Borstel et al. 2003b], and show its translation into detailed UML diagrams to depict components, interrelationships, and procedures using the object-oriented paradigm [Borstel and Gordillo 2004b]. Later sections show a general teleoperation architecture, as well as control design requirements to build VLs.

3.1 Virtual Laboratory Concept

In the literature, the VL concept is defined as a heterogeneously distributed environment that enables collaboration between geographically separated groups of researchers to work on common projects [Consortium 2004], or as a system that allows users to plan and conduct experiments and collect and analyze data by providing teleimmersion mechanisms [Guimaraes et al. 2003a]. In general, two types of VLs for telerobotics are defined:

Simulated: These laboratories give access to simulated equipment and its surrounding environment through a computer connected to the Internet. The simulated
equipment allows performing tests or completing experiments, giving unlimited access to users and protecting the equipment from unskilled users.

**Remote:** Physical equipment inside a laboratory facility is remotely accessed through a computer connected to the Internet. Remote experimentation is possible using this kind of laboratory. Observation of the experiment is done by using information from sensors connected to the Internet, which are placed in the laboratory facility. The user interacts with real equipment, facing practical situations that could be difficult to model in a simulation. In certain implementations, simulation is included to enhance user interaction with the experiment.

![Simulated Virtual Laboratory](image1)

![Remote Virtual Laboratory](image2)

Figure 3.1: Simulated and remote Virtual Laboratories.

The necessary functionalities of VL systems are closely related to the following definition:

A *true experiment* is defined as a study in which certain independent variables are manipulated, their effect on one or more dependent variables is determined, and the levels of these independent variables are assigned randomly to the experimental units in the study [Hicks 1982].

Experiments performed in VLs are developed in distributed and heterogeneous laboratory environments providing a high level of control. The objective is to manipulate one or more independent variables and control all other specific independent variables at fixed levels. The effect on dependent variables caused by manipulation is measured. Result analysis can be performed, based on the effect measurement [Wohlin *et al.* 2000]. Thus, manipulation and randomization are necessary to perform *true experiments* in VLs from which the user may be able to infer cause and effect.
To explain the true experiment concept, visualize a simple pendulum experiment: a pendulum is any mass that swings back and forth on a rope, string, or chain. If the mass is pulled away from its resting position, so that the string is at an angle, and the mass is then released, the mass will begin to swing back and forth. The length of time it takes the mass to swing one time all the way over and back is called the period of the pendulum.

Three experiments can be performed to deduce if the amount of mass, the angle of release, or the length of the string affect the period of the pendulum [Willis 2005]. In these experiments, the independent variables manipulated to mount each experiment are: the attachment point of the string, the string itself, the method used to time the pendulum, and any other variable which is not currently tested. These variables remain the same for each test, so they will not affect the experimental results.

Tested specific independent variables are defined at fixed levels in each experiment. These are the mass, the angle, and the length of the string. For example, to test if the amount of mass affects the period, the angle and the length are not manipulated during the test. Starting with a certain amount of mass, the experiment is executed by releasing the mass, then the dependent variable is measured; this is the period. After that, the amount of mass is increased at a fixed level, executing the experiment every time the mass is increased, and measuring the period. Experimental results allow the user to infer that mass variation has no effect on the period, and the angle variation has no visible effect on the period. However, the length variation of the string produces a direct effect on the period [Halliday and Resnick 1999].

This simple experiment describes the necessary functionalities to perform a true experiment: mount, define, and execute. Therefore, a VL system for telerobotics must have these functionalities to remotely perform a true experiment:

Mount: A VL must allow the user to introduce and manipulate independent variables to set up the simulated or real experiment.

Define: A VL must provide a way to determine specific independent variables grouped at fixed levels.

Execute: A VL must allow an experiment to run to infer cause and effect on dependent variables based on the effect measurement at fixed levels.

In this study, a VL for telerobotics is defined as a system that provides access to one or more users to operate laboratory equipment in a simulated or remote way to mount, define, and execute experiments. These systems provide capabilities to observe, supervise, and interact with the experiment.
3.2 Model Elements

The following essential elements allow the proposed model to perform robotic teleoperation over the Internet:

*Guest*: Computer-based system containing user interface(s) that allows the user to observe, mount, and execute experiments in a remote or simulated way.

*Host*: Computer-based system that performs tasks requested by the user, providing access to simulated or physical equipment, depending on the VL type.

*Media*: Computer digital networks used to establish the information flow between *Guest* and *Host*. In terms of the conceptual layer model OSI (Open System Interconnection), *Media* includes low physical communication layers to higher transport protocols, such as the TCP/IP (Transport Communication Protocol/Internet Protocol) suite.

Figure 3.2 describes the model elements and the most significant components needed to perform remote experimentation.

![Figure 3.2: Virtual Laboratory elements and the most significant components.](image)

The VL model is composed of at least a single *Guest* element connected to a single *Host*; nevertheless, other possible configurations can be envisioned, such as, a single *Guest* connected to multiple *Hosts*, multiple *Guests* are connected to a single *Host*. The model takes into account the possibility of sharing a resource with multiple users, such as using task queues or round robin management, or using multiple resources by one user. Furthermore, it allows interaction between multiple *Guests* and multiple *Hosts*. The model leads to an implicit modularity, which is used for VL development and creation of new capabilities in implemented VLs.
3.3 Abstract Model

The abstract VL model represents processes enclosed by computer-based systems (Guest and Host), which are required to perform telerobotics experimentation over the Internet. The model considers generic processes to build a VL for telerobotics and can be customized by either including or discarding these processes depending on experimental specifications. Modularity means that the model allows its repetitive use to develop VL functionalities or increase capabilities in VL implementations.

![Diagram of abstract model](image)

Figure 3.3: Abstract model to develop VLs for telerobotics over the Internet.

Guest system hardware is composed of a computer containing Guest software; the computer is connected to Input Devices (e.g. pointer devices, joysticks, keyboards, and others), and Output Devices (Displays, Effectors, or Transducers) to feed the user with sensory information.

When the Guest system is working, Input Devices acquire analog actions from the user. These actions are converted into discrete inputs by the Input Devices; then these inputs are interpreted by procedures enclosed by an Input Processing entity, which interacts with other entities to perform user requests. The Input Processing can send commands to a Simulation entity. In contrast, a Sensory Conversion entity processes sensory information from Host and Simulation. This Sensory Conversion entity allows
a complete perception of the experiment by transforming sensed and simulated data into device signals to activate Output Devices.

The Simulation entity in Guest represents procedures to provide immediate feedback. In simulated VLs, these procedures access Host to retrieve Software Libraries containing simulated models of laboratory equipment and its surrounding environment. In remote VL systems, Simulation is a tool to foresee the effects of commands applied on the experiment. For enhanced reality feedback, Simulation is used to synchronize simulated equipment with updated information (specific data) from the Host provided by a Sensed Data Processing entity. The Task and Settings Database entity represents procedures to establish a database in Guest to keep tasks descriptions and configuration settings to recreate performed experiments.

Host hardware is composed of a computer connected to Sensor devices (e.g. cameras, electromechanical transducers, and others), and Mechanism and Control Devices to perform robotic experimentation in the Remote Environment.

When a remote VL system is working, an Experiment Control entity regulates the experiment behavior. This entity encapsulates control algorithms belonging to specific experiments and procedures to convert control commands into required signals for Mechanism and Control Devices. The control algorithms can work in an open-loop control mode using user commands sent by the Input Processing as reference inputs or in a closed-loop control mode receiving specific data from Data Sensed Processing as a feedback loop. The Data Sensed Processing entity encloses procedures, which acquire sensory information from Sensors. These procedures convert the sensory information into discrete data to be analyzed and generate specific data. Sensed data is sent to Guest as a sensory feedback. Sensor devices are located in the Remote Environment to capture sensory information from the experiment. Experimental data is stored by procedures enclosed by the Experiment Database entity. Mechanism and Control Devices are robotic devices with electromechanical components to develop the experiment.

A Communication entity includes procedures for both managing and sharing information between the model elements (Guest and Host) over Media (Internet), using one or several communication technologies to ensure its transportation through the Internet.

### 3.4 Object-Oriented Modeling

The VL abstract model is translated into a modeling formalism to describe its intrinsic components defining a general framework that will be used by a development methodology to design VL applications. UML is an object-oriented language that captures the idea behind the VL model. It is a standard language to generate software
blueprints used to visualize, specify, construct, and document computer-based systems. It has three goals. First, model systems from concept to executable artifacts. Second, address the issues of scale inherent in complex systems. Third, create a modeling language usable by humans and machines [Booch et al. 1999]. UML is composed of several graphic elements and rules that are combined into diagrams known as a model. These diagrams give different perspectives of complex software systems [Schmuller 1999]. In terms of the views of a model, UML defines the following graphic diagrams:

- Use case diagram
- Class diagram
- Behavior diagrams:
  - Statechart diagram
  - Activity diagram
- Interaction diagrams:
  - Sequence diagram
  - Collaboration diagram
- Implementation diagrams:
  - Component diagram
  - Deployment diagram

The best models are those that let the developer choose the degree of detail, which will depend on who implements the design and which specifications need to be reviewed. Therefore, UML diagrams may be expressed at different levels of precision. Moreover, it is best to have models that are clearly connected to reality. Since models simplify reality, the designer has to make sure that these simplifications do not mask important details. Furthermore, complex software systems are better approached through a small set of nearly independent diagrams; no single diagram is sufficient [Fowler 2002]. Thus, several complementary and interlocking views are needed to understand the complexity of a VL system from an object-oriented perspective:

Class diagrams are the most common diagrams found in modeling object-oriented systems. They address the static design view of such systems, showing a set of objects and their relationships.

Sequence and collaboration diagrams are kinds of interaction diagrams. Interaction diagrams address the dynamic view of a system. A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages; a collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
Deployment diagrams show the configuration of run-time processing nodes and the components that live on them. They address the static deployment view of VL application architectures.

Use case diagrams show a set of use cases and actors (users) and their relationships to address the static use case view of the VL system. These diagrams are especially important in organizing and modeling behavior of a system.

Statechart diagrams address the detailed dynamic view of a system, which are represented by state machine automata. Statechart diagrams are especially important in modeling the behavior of VL systems. Statecharts allow their transformation into another dynamic modeling notation (Petri Nets) to analyze and validate the behavioral design of event-based VLs.

3.4.1 Virtual Laboratory Classes

Components of the abstract VL model are defined in the UML formalism by the class diagram. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. VLs for telerobotics can be considered as a class or set of objects with operation methods to mount, define, and execute telerobotic experiments. In general, the VL class has two subclasses (Simulated and Remote), which inherit VL class attributes and operation methods. This definition is represented by UML class diagrams as shown in Figure 3.4, where the rectangle on top represents the VL class, which includes its name, attributes, and operation methods, respectively. VL subclasses are rectangles with similar attributes and operation methods. These subclasses are connected to the VL class by lines with a non-filled triangular symbol indicating the main class.

![UML Class diagram representing the VL class and its subclasses: Simulated and Remote.](image)

Figure 3.4: UML Class diagram representing the VL class and its subclasses: Simulated and Remote.
3.4.2 Simulated Virtual Laboratory

The goals of a Simulated VL are mount, define, and execute simulated robotic experiments in Guest. Simulated experimentation is performed by retrieving, via the Internet, software libraries stored in Host. These software libraries model mechanisms and control devices with their surrounding environment and are written in programming languages for visual modeling, such as VRML or using graphic libraries for programming languages, such as OpenGL (Open Graphic Library for Visual C++). Class components of simulated VLS represent abstract or physical entities, that is software applications or electromechanical devices. Figure 3.5 describes the subclasses, which are part of the simulated VL class.

![UML Class diagram to represent the Simulated VL class. This figure represents the most significant components in a simulated VL class.](image)

A simulated VL is composed of three different subclasses: Guest, Media, and Host, representing the previously mentioned VL elements. These subclasses are defined as components of the Simulated VL class. They are conceptually associated by using lines and filled rhombic symbols. Conceptual associations have numbers and symbols to indicate their multiplicity, that is, Guest system relates to Media in a relation of one or more (1..*) to one (1), respectively.

In Figure 3.5, the Guest system has a Guest software subclass, which is described in detail in a later section. The output device class represents physical devices for sensory feedback, such as displays, transducers, and effectors, while the computer class represents the Guest computer in the abstract VL model. The computer class is composed necessarily of a network board class to communicate Guest and Host systems via Media (Internet). Notice that several components of the computer class were not represented in Figure 3.5 to keep the model simple. Media is composed of another
three subclasses to encapsulate network devices (routers, hubs, network converters, among others), gateways (computers connecting different networks), and physical network (cables, optic fiber, connectors, among others). The Host system is composed of computer, software, and model library subclasses. Computer class is composed of a network board subclass. Host software is described in a later section. The model library class is composed of two abstract subclasses describing mechanism and control devices and remote environment models, which simulate laboratory equipment and their surrounding environment. Appendix A contains a detailed description of Simulated VL subclasses.

### 3.4.3 Remote Virtual Laboratory

The goals of a Remote VL are mount, define, and execute remote experiments from Guest. Remote experimentation is performed by teleoperating robotic equipment inside a laboratory facility via the Host system through the Internet. Figure 3.6 shows a class diagram describing the most significant components of a remote VL class.

![Figure 3.6: UML Class diagram representing most significant components of remote VL class.](image)

The class diagram specifies that the remote VL class is composed of the same elements described by the VL concept. In remote VLs, only the Host system changes its class composition. The Host system is composed of several subclasses for encapsulating remote environment, mechanism and control devices, computer, Host software, and sensor devices. The remote environment class represents physical workspace. Mechanism and control devices describe robotized physical devices. The computer class represents the Host computer in the abstract VL model. The sensor class represents physical sensor devices located in the remote environment. The Host software is described in detail in the next section.
3.4.4 **Guest and Host Software**

The software class description for Guest and Host is shown in Figure 3.7. Guest software is composed of classes representing generic procedures in Guest to allow the user to communicate with Host and interact with the experiment.

![UML Class Diagram](image)

**Figure 3.7:** Guest and Host software definition using the UML class diagram. Software classes and their operation methods are described in detail in Appendix A.

**Guest software** is composed of five subclasses:

**Input Processing:** Input interpretation, comparison with simulated data, saving, and retrieving configuration settings or tasks are the operation methods performed in the Input Processing class.

**Sensory Conversion:** Sensed and simulated data are restored and converted into appropriate device signals to provide the user with sensory information.

**Simulation:** This class retrieves and parses the model definition to execute the simulated model and provide simulated data to Sensory Conversion.

**Settings/Tasks Database:** Searching, reading, and writing are basic operation methods to save and retrieve useful information regarding the experiment settings and performed tasks.

**Communication:** Represents transmission and reception management of data via Media.

**Host software** is composed of several subclasses to perform simulated or remote robotic experimentation. The following classes compose Host software:
Sensed Data Processing: Performs data acquisition, data reduction, and analysis to obtain specific information regarding the experiment.

Model Library: Retrieves model libraries requested by Simulation and stores libraries containing model definitions regarding control methods, device specifications, interaction procedures, and environment specifications.

Experiment Database: Searching, reading, and writing are basic operation methods of this class to retrieve or save experimental data.

Experiment Control: Performs command interpretation to decide which ones are settings for control algorithms or which ones are commands for Mechanism and Control Devices. It executes control algorithms using settings issued by the user and verifies and translates control outputs and commands into appropriate signals for Mechanism and Control Devices.

Communication: This class represents transmission and reception management of data via the Media.

Guest and Host software are associated by their communication components to Media class. Class components integrating Guest and Host software are described in detail in Appendix A.

3.4.5 Software Collaboration Diagram

The previously described Guest and Host software components have relationships between them. These relationships are described in an UML collaboration diagram shown in Figure 3.8. This collaboration diagram shows messages sent by class components of the VL framework.

In UML collaboration diagrams, arrows indicate message direction. In Figure 3.8, the messages are data flowing between instantiations of VL class components representing general procedures and physical devices, that is, the sensory conversion instantiation interacts with communication, simulation, and output device instantiations. The sensory conversion receives sensed data from the sensed data processing through the communication instantiation, then the received sensed data is converted into device signals for output devices. Sensory conversion also receives simulated data that is converted into device signals for output devices.

3.4.6 Deployment Diagram

The UML deployment diagram defines how VL model classes should be distributed into physical hardware to implement a VL application. Figure 3.9 shows a general VL deployment diagram. Computers and physical devices are represented as three-dimensional boxes. Guest and Host computers encapsulate software components
3.4. Object-Oriented Modeling

Figure 3.8: UML collaboration diagram of VL components. It defines relationships and data flow among several instantiations to integrate a VL system.

that are represented as boxes with two rectangles at its left side. Figure 3.9 shows that computer Operating System (OS) and Guest software are located in the Guest computer. In a similar way, OS and Host software are encapsulated in the Host computer. Dotted arrows indicate that Guest and Host software are OS dependent.

Internet (Media) is symbolized as a cloud with conceptual associations to both computers. These associations are lines representing connections, such as physical cables, satellital links, wireless connections, and infrared links, among others. Guest and Host computers are connected through the Internet physical medium and communicated by the data transport protocol suite (TCP/IP). Physical devices are also connected with Guest and Host computers: input and output devices are connected to the Guest computer, and sensors and mechanism and control devices are connected to the Host computer. Connection relations are labeled using customized stereotypes to describe data passing through them, that is <<device signal>>.

3.4.7 Use Case Diagram

Use case, sequence, and statechart diagrams are used to capture the dynamic aspects of VL systems. Use case diagrams represent the intended behavior for VLs without having to specify how that behavior is implemented and to provide a way for developers to come to a common understanding with the system’s end user and domain experts.

UML use case diagrams describe the expected functionalities of a VL system from
Chapter 3. Virtual Laboratory Modeling

Figure 3.9: UML deployment diagram of the VL. It defines how the several objects that integrate the system are physically distributed and connected.

In this study, VLs allow the user to mount, define, and execute simulated or real experiments at a remote site. Therefore, a VL system for telerobotics has three use cases: Mount, Define, and Execute.

The UML use case diagram in Figure 3.10 describes the listed use cases. This UML use case diagram defines three main functionalities for VL systems (Mount, Define, and Execute). The Mount use case is extended to the Define use case, then the Define use case is extended to the Execute use case. These functionalities are performed by three subsystems sharing common objects, operation methods, and attributes needed to perform a true experiment. The diagram also describes how the first subsystem is expanded to the second and the third by discarding or adding new components. Each subsystem typically involves a repetitive use of VL classes. Therefore, this work considers each subsystem as a customized instance of the UML VL framework. Detailed aspects for combining and providing a synchronized execution of each subsystem (VL framework instantiations) are provided in the proposed development methodology in Chapter 4.

Figure 3.10: UML use case diagram of the VL framework defines three functionalities for VLs: Mount, Define, and Execute.
3.4.8 Sequence Diagram

A sequence diagram is an interaction diagram that emphasizes the time ordering of messages inside a VL system. Graphically, it is a table that shows object instantiations arranged along the x axis and messages, ordered in increased time, along the y axis. Vertical dotted lines underneath object instantiations are their time lines. Vertical bars on time lines represent the instantiation lifetime and horizontal arrows represent messages sent among instantiations.

The sequence diagram captures the detailed interaction between the user and VL subsystems. Figure 3.11 shows a general UML sequence diagram of VLs for telerobotics, where the subsystems are represented by the Mount, Define, and Execute instantiations. The diagram defines that these subsystems are executed in a sequential and synchronized way, and the user receives feedback to know the status of each experiment stage.

Notice that during the execution of the experiment, the user is allowed to introduce commands to modify specific independent variables. However, these specific variables are not necessarily tested in the experiment execution. For instance, when the user mounts, defines, and executes a robotic teleoperation experiment, the independent variables can be the type of sensors to be used, the robot itself, the method used to drive the robot, or the robot workspace. Specific independent variables to be defined and tested can be the sensors’ range or the scalar constants to tune-up a PID controller for robot effectors.

Specific independent variables, introduced during the experiment execution stage, are the commands issued by the user from Guest, which are necessary to produce dependent variables, such as sensor measurements, position, or speed. Since each introduced specific variable (control command) generates a new dependent variable (system output), the user has to receive an updated feedback from the experiment. Therefore, robotic teleoperation increases the interaction between Guest and Host systems.

3.4.9 Statechart Diagram

Statechart diagrams are used to describe the behavior of instances of a model element, such as an object or an interaction. Specifically, statecharts describe possible sequences of states and actions through which the element instances can proceed during its lifetime as a result of reacting to discrete events, such as signals and operation invocations.

A UML statechart diagram is one way to describe and specify internal changes in VL subsystems. When an object is modified because of internal events in the system or time, then the object has changed its state. Statechart diagrams show the
inner states of objects, transitions between states, and indicate a start and finish of consecutive state changes.

Graphically, rounded rectangles represent object states; transitions between states are represented by arrows connecting states, a filled circle represents the start and another circle with a filled dot represents the final state. A general statechart diagram for VLs is shown in Figure 3.12. It shows inner states of mount, define, and execute VL subsystems.

The statechart depicted in Figure 3.12 describes, in a general way, the expected behavior of a VL system. For example, once the VL is started, it executes a procedure to acquire experiment settings. When the user introduces experiment settings, the VL system goes to the next state and immediately generates sensory feedback regarding the introduced settings. The VL outputs sensory information via its output devices to be received by the user.

Then, a procedure to acquire setting confirmation is executed. If the settings are not confirmed (not correct), then the setting acquisition procedure starts again. If the settings are confirmed, then the system goes to the confirmed setting state and is ready to mount the experiment. This state-transition sequence continues until the user stops the VL system.
Figure 3.12: General UML statechart diagram of the VL framework that defines internal states of mount, define, and execute VL subsystems.
3.4.10 UML Statechart Diagram Formal Definition

A statechart diagram is a representation of a state machine (finite automaton) emphasizing the flow of control from state to state. Statecharts are graphs of states connected by transitions; transitions indicate movements from one state to another. Statechart diagrams represent the behavior of entities capable of dynamic behavior by specifying their response to the receipt of event instances. Typically, this diagram is used for describing the behavior of class instances, but statechart diagrams may also describe the behavior of other entities such as: use-cases, actors, subsystems, operations, or methods [Booch et al. 1999].

Let $ST$ denote a statechart diagram; $ST$ is defined by the following seven-tuple [Jansamak and Surarerks 2004]:

$$ST = (S, TST, E, C, A, i, f),$$

where:

- $S$ is a finite set of states.
- $E$ is a finite set of events.
- $C$ is a finite set of conditions.
- $A$ is a finite set of actions.
- $TST \subseteq \{S \times E \times C \times 2^A \times S\}$ is a finite set of transitions where $2^A$ denotes the power set of $A$.
- $i \in S$ is a initial state.
- $f \in S$ is a final state.

The following remarks are concerning the statechart diagram definition.

In statecharts, each transition arrow has a label that comes in three parts: Event/Condition/Activity; all the parts are optional. The occurrence of an event $v$ fires a transition $t$ ($t \in TST$) if the machine is in the source state $s$ ($s \in S$) of $t$; the occurrence of $v$ matches event $e$ ($e \in E$) of $t$, and condition $c$ ($c \in C$) of $t$ holds. Since label parts are optional, a missing activity $a$ indicates that the machine does not do anything during transition $t$. A missing condition $c$ indicates that the machine always takes transition $t$ if the appropriate event $e$ occurs. A missing event $e$ indicates that the machine take the transition $t$ immediately, which is seen mostly with diagrams representing activities-states [Fowler 2002].

In this study, transitions are labeled with activities $a$ that usually process user inputs, e.g. “acquire experiment settings” where the activity description (acquire)
includes the user input (experiment settings); otherwise, transitions are taken immediately by the machine, e.g. "mount experiment", where the activity (mount) needs the previous state as input. Therefore, the statechart diagram formal definition is simplified to the following five-tuple:

\[ ST = (S,T_{ST},A,i,f) \]  

(3.2)

On the other hand, a state machine is an automaton capable of representing a language according to well-defined rules. The set of labels for the transitions is the event set (alphabet \( \Sigma \)) of the automaton. Given the finite set \( \Sigma_{ST} \) (the alphabet of statechart \( ST \)), notation \( \Sigma_{ST}^{*} \) denotes the set of all finite sequences (strings) of elements in \( \Sigma_{ST} \), including the zero length sequence \( \epsilon \).

Notation \( f(y, a) = x \) means that if the automaton is in state \( y \), then upon the occurrence of event \( a \), the automaton will make an instantaneous transition to state \( x \). This notation is called the transition function, while it is extended to \( f(y, a) = x \), where \( a \in A^{*} \). On the other hand, notation \( \Gamma(x) \) is the set of all events \( a \) for which \( f(x, a) \) is defined and called the active event set (feasible event set) of the automaton at state \( x \) [Cassandras and Lafortune 2001].

**Definition 1:** Let \( L(ST) \) denote the language generated by \( ST \), defined as

\[ L(ST) = \{ w \in \Sigma_{ST}^{*} \mid \exists y \in S, \text{ such that } \Gamma(y) = w \text{ and } f(i,w) \text{ is defined} \}. \]  

(3.3)

### 3.5 Event-based Architecture

This work also introduces a general architecture that provides foundations for required stability, synchronization, and autonomy properties in designed VLs, and supplies the necessary control design for VL development. This architecture combines event-based and supervisory control approaches. Therefore, the architecture is non-time referenced; it can provide a closed-loop control; it takes into account prediction; it can be customized to fulfill specifications of each VL subsystem (mount, define, and execute), and it is compatible with Petri Net modeling techniques.

#### 3.5.1 Event-Based Planning and Control Theory

The major difference between transferring information and performing bilateral teleoperation via the Internet is that the latter requires action synchronization that is crucial for the success of execution of the actions [Xi and Tarn 1999]. Since packet exchange in the Internet is affected by the packet’s routes and handling policies at each node the packet traverses, the communication time delay is a random variable [Oboe and Fiorini 1997, Bolot et al. 1990]. This random delay has resulted in the loss of synchronization of time-based action references among components of bilateral teleoperation systems, causing their instability [Xi and Tarn 1999].
Event-based control systems are immune to almost all the delay-related problems because they are non-time referenced systems. To achieve stability and synchronization over the Internet, VL subsystems are designed as event-based, action-referenced systems, which are developed based on the *event-based planning and control theory* discussed by [Xi 1993].

Signals in VL subsystems must be sampled with respect to an action reference other than time. Equivalently, this implies that system components reference a non-time based action reference. The idea is to model the system and task as a function of a non-time-based variable, which is denoted by \( s \) and called the *event-based action reference*. Figure 3.13 shows the comparison between the typical time-based and the event-based control schemes [Elhajj et al. 2003].

In the time-based control scheme depicted by Figure 3.13(a), the Planner block represents a planner process (e.g. operator) that introduces a reference input \( y^d \) (e.g. command) into the system. This reference \( y^d \) is compared with the system's output \( y \) received via the feedback loop to produce error \( e \). This error \( e \) is introduced into the Controller block to regulate Robot behavior. Variable \( d \) represents disturbances affecting Robot behavior.

In Figure 3.13(b), the Action Reference block acts like a clock for the system. It is a map from the output or state of the robotic system \( y \) to a scalar variable \( s \). Formally, it maps the output of a robotic system \( y \in Y \) to a scalar variable \( s \in S \),

\[
\Pi : Y \rightarrow S. \tag{3.4}
\]

Reference \( s \) is usually taken to be a physical output of the system, such as distance or position. For example, \( s \) can be chosen as the distance traveled by the end effector of the Robot. However, the event-based reference \( s \) can also be a virtual one that does not correspond to any physical quantity. For example, \( s \) can be chosen to be the number of control cycles (iterations) the system has performed.

The design criterion for the event-based stability is based on the following theorem that was introduced and proved in [Xi 1993, Tarn et al. 1996, Xi and Tarn 1999, Tan et al. 1999].

**Theorem 1:** If the original robot dynamic system (without remote human/autonomous controller) is asymptotically stable with time \( t \) as its action reference; and the new non-time action reference, \( s = \Pi(y) \) is a (monotone increasing) nondecreasing function of time \( t \), then the system is (asymptotically) stable with respect to the new action reference \( s \) [Xi 1993].

This theorem was proved by using the Converse Theorem [Hahn 1967] in [Xi 1993, Xi and Tarn 1999]. The above theorem provided a way to develop stable event-based VL systems over the Internet. The only requirement is that the robot
must be a stable system, meaning that robot dynamics is asymptotically stable with time $t$ as its action reference.

For teleoperation systems, some considerations have to be made to apply this approach. First, in teleoperation, there is no predefined path. The path is generated based on received feedback. Second, the system should be transparent to the operator. Hence, feedback should accord with the current state of the system. Therefore, the system has to be designed to eliminate the buffering effect of delay.

The buffering effect is caused by many delayed signal instances flowing within the network. This effect provokes the user to make decisions and introduces new commands into the system based on signal instances that reflect past states of the system. Then, the system becomes unstable, loses its transparency, and suffers desynchronization.

A great advantage of the event-based control approach is its ability of translating its dynamics into Petri Net formalism, and providing design techniques based on behavioral properties of Petri Nets to analyze and assure stability and synchronization of designed teleoperation systems [Elhajj et al. 2001, Elhajj et al. 2003, Elhajj et al. 2004].
3.5.2 Event-Based Design Requirements

The developer must take into account two necessary properties of event-based systems to produce VL design procedures. When the $s$ reference variable is chosen, it has to be chosen in a way that makes the system event-synchronized [Elhajj et al. 2002a, Elhajj et al. 2004] and the event-based sampling period $S$ has to satisfy the following sampling condition [Unser 2000, Elhajj 2002]:

$$\frac{\pi}{KS} \geq w_{max}, \quad (3.5)$$

where $K$ is a scaling constant between event-based reference and time and the sensed sensory feedback is band-limited by the maximum angular frequency $w_{max}$ and $f_{max} = w_{max}/\pi$ is its Nyquist frequency.

The event-syncronization property means that system signals (control and feedback) are always referencing the same event-based reference. An event-synchronized teleoperation system is one in which all the signals (control and feedback) in the system are always referencing the same event-based reference [Elhajj et al. 2003]. Event-synchronization definition is similar to time synchronization for open loop systems. However, event-based applications for bilateral teleoperation are closed-loop systems. Instead of using time as reference, the event-based reference is being used, and it includes the control signal, too.

Furthermore, when different sensory informations are sensed by a teleoperation system, a master sensory information would be chosen and synchronization would be carried out with respect to it. For example, with video and force, force can be the master stream (provided in real time) since it is more informative and requires less bandwidth than video [Elhajj 2002].

3.5.3 Petri Net Formal Definition

A Petri Net is a graphical and mathematical modeling tool. Several Petri Net behavioral properties are used to analyze VL dynamics and verify VL event-synchronization. These models were first developed by [Petri 1962]. Transitions, places, and arcs characterize a Petri Net graph. Such a graph has two types of nodes, places and transitions; arcs indicate the relationships between places and transitions. Petri Nets are bipartite graphs since arcs cannot directly connect nodes of the same type. Rather, arcs connect places to transitions and transitions to places. Several conditions have to be satisfied so that a transition may occur.

Petri Net notation is associated with many types of systems in the engineering field, such as manufacturing, telecommunication, and robotics. A Petri Net advantage is its ability to decompose or modularize a potentially complex system.
3.5. Event-based Architecture

[Cassandras and Laafortune 2001].

In graphical representation, places are drawn as circles, transitions as bars or boxes. Arcs are labeled with their weights (positive integers), where a $k$-weighted arc can be interpreted as the set of $k$ parallel arcs. Labels for unity weight are usually omitted. A marking (state) $M$ assigns a nonnegative integer $k$ to each place $p$. $M$ is an $m$-vector where $m$ is the total number of places $p$. The $i$th component of $M$, denoted by $\mu(p_i)$, is the number of tokens in place $p_i$.

$$M = \mu(p_1), \mu(p_2), \mu(p_3), \ldots \mu(p_m)$$  \hspace{1cm} (3.6)

It is said that a place $p_i$ is marked with $k$ tokens (denoted as $\mu(p_i) = k$). Pictorially, $k$ black dots (tokens) are in $p_i$. A simple Petri Net is shown in Figure 3.14.

![Figure 3.14: Simple Petri Net with two places, two transitions and one token in $p_1$ ($\mu(p_1) = 1$); thus, its marking is $M = 1, 0$.](image)

Formally speaking, let $PN$ denote a Petri Net. It is defined to be the five-tuple:

$$PN = (P, T_{PN}, \xi, M_o, \alpha)$$  \hspace{1cm} (3.7)

$$P \cap T_{PN} = \emptyset \text{ and } T_{PN} \cup P \neq \emptyset,$$  \hspace{1cm} (3.8)

where $P$ denotes the finite set of places of $PN$, $T_{PN}$ is the finite set of transitions of $PN$, $\xi \subseteq \{P \times T\} \cup \{T \times P\}$ denotes the finite set of directed arcs connecting places and transitions of $PN$, $M_o$ denotes the initial marking of $PN$, and $\alpha \in \Sigma_{PN}$ is the labeling function that associates with each transition $t \in T$, an event $\alpha(t)$ belonging to the finite event set $\Sigma_{PN}$. The labeling function $\alpha$ is extended to $\alpha = T_{PN}^* \rightarrow \Sigma_{PN}^*$ in a natural way [Kumar and Holloway 1996].

In the Petri Net formalism, for each $t \in T_{PN}$, notation $t^o$ and $t^o$ are used to denote the set of "output" and "input" places of $t$, respectively, i.e.,

$$t^o = \{p \in P|(t, p) \in \xi\},$$  \hspace{1cm} (3.9)

$$t^o = \{p \in P|(p, t) \in \xi\}.$$  \hspace{1cm} (3.10)
Similarly, for each \( p \in P \), \( p^o \) and \( p^o \) denote the sets of transitions for which \( p \) is an “output” and “input” place, respectively, i.e.,

\[
p^o = \{ t \in T | (p, t) \in \xi \},
\]

\[
p^o = \{ t \in T | (t, p) \in \xi \}.
\]

The number of tokens may change during the execution of \( PN \), where transitions are active components. The change of state of \( PN \) is according to the following firing rule [VanDerAalst 1998]:

1. A transition \( t \in T_{PN} \) is said to be enabled if, and only if, each input place \( \delta t \) contains at least one token.

2. An enabled transition may fire. If transition \( t \) fires, then \( t \) consumes one token from each input place \( \delta t \) and produces one token in each output place \( t^o \).

Given a marking \( M \) and \( r \in T_{PN}^* \), notation \( M[r > M'] \) is used to denote the marking reached \( M' \) by firing the transition sequence \( r \) starting at \( M \), provided \( r \) is fireable in \( M \), else it is undefined [Kumar and Holloway 1996].

Letting \( L(PN) \) denote the language generated by \( PN \), it is defined as:

\[
L(PN) = \{ z \in \Sigma_{PN}^* | \exists \tau \in T_{PN}^*, \text{ such that } \alpha(\tau) = z \text{ and } M_o[\tau > \text{ is defined}] \}. (3.13)
\]

The language generated by Petri Nets is referred to as \( P \)-type language. In a Petri Net, the occurrence of a language symbol can be represented by the firing of a transition [Moody and Ansalklis 1998].

### 3.5.4 Proposed Architecture

The event-based control approach and some of its considerations can be applied to VL systems for telerobotic experimentation. To mount, define, and execute experiments, VL subsystems have to be performed in a sequential and synchronized way. Signal flow between Guest and Host must correspond to the current experimental task and its state. An event-based design for VL subsystems must take into account a reference parameter that reflects subsystem behavior, such as a measure of its effect on the remote environment (distance, and position, among others). A teleoperation architecture that considers the event-based approach and the VL model entities is shown in Figure 3.15.

The proposed architecture is composed of the model entities previously described in the abstract VL model. The Mechanism and Control Device block encapsulates the effectors and mechanisms composing a typical robotic device. Where \( d(t) \) represents disturbance introduced by the Remote Environment, \( y(t) \) is the output of the VL system, and \( e(s) \) is the error resulting from the comparison of \( y'(t) \) from the Sensed
3.5. Event-based Architecture

Figure 3.15: General teleoperation architecture to provide control design in VL development. This architecture relates the event-based control with the VL model entities.

Data Processing and Sensor feedback loop and signal reference $y_s(s)$. The system effect $x(t)$ resulting from the last $y(t)$ is acquired by the Sensor device and Sensed Data Processing blocks to supply the non-time based action reference $s$ to the Sensory Conversion block. The $s$ reference synchronizes the experiment state with the sensory signal $q(s)$ generated by the Sensory Conversion and Output Device blocks. Then, the Operator using the Input Device and Input Processing blocks produce the reference input $y_s(s)$ that is used by the Experiment Control block.

For the Simulation block, $s$ is also the synchronization reference to generate a predicted signal $y_p(s)$ starting from the last $y_s(s)$. This means that Simulation is synchronized to feedback sensory information, based on the last Operator’s command. Therefore, it can generate useful sensory information in real-time or predict command effects and robot status in the Remote Environment.

From the Supervisory Control point of view, Experiment Control can provide stability and autonomy to carry on planned tasks requested by the Operator. In that case, the control blocks in the Operator’s side behave as the planner block of Figure 3.13, where the Operator is considered as a supervisor, then $y_s(s)$ is treated as a high level command and the Experiment Control output is the low level command to control the robot.
The stability of the proposed teleoperation architecture can be formally stated with the following corollary of Theorem 1:

**Corollary:** If the dynamics of the system closed by the Experiment Control and the Sensed Data Processing-Sensor blocks, with time $t$ as its action reference, is asymptotically stable, then the control loop closed by the Operator and the Remote Environment, using an action reference $s$ (non-decreasing function of time $t$), is asymptotically stable with respect to $s$.

**Proof:**

**Necessary:** Assume that the dynamics of the system closed by the Experiment Control and the Sensed Data Processing-Sensor blocks, with time $t$ as its action reference, is unstable. It means that the robot control dynamic system (without the human Operator) is unstable. Therefore, it does not satisfy Theorem 1 requirements to design a stable event-based teleoperation system [Elhajj et al. 2000b].

**Sufficient:** Now, assume that the dynamics of the system closed by the Experiment Control and the Sensed Data Processing-Sensor blocks, with time $t$ as its action reference, is asymptotically stable. This means that the robot control dynamic system is asymptotically stable, allowing the use of Theorem 1 as a design criterion [Elhajj et al. 2000b].

Since Sensed Data Processing-Sensor blocks in the control loop closed by the Operator and the Remote Environment act as the Action Reference Block in the control scheme shown in Figure 3.13(b). Scalar variable $s$ maps the output or state of the robotic system in the Remote Environment $x(t)$; thus, the reference $s$ is taken to be a physical output of the system (distance to the origin, angle, absolute position, among others). Then assume that the reference $s$ is chosen in a way that satisfies Eq. 3.15. This implies that the system satisfies event-based design requirements [Elhajj 2002, Elhajj et al. 2003].

Furthermore, control and feedback signals in the Sensory Conversion-Output Device, Input Device-Input Processing, and Simulation blocks are always referenced to the same common event $s$ at any point in time. This implies that the Operator-Remote Environment closed loop is event-synchronized. This property implies that two different control blocks cannot be at different events; in this closed-loop, the feedback obtained by the Operator has to correspond to the most up-to-date status of the robot being controlled, or corresponds to a simulated prediction synchronized with the most up-to-date status of the robot. Therefore, control signals have the same action reference stamp as the feedback being rendered. Thus, the Operator and the Robot are always synchronized in event regardless of time delay and its variations [Elhajj et al. 2000a].
Finally, assume that the non-time action reference $s$ is a (monotone increasing) non-decreasing function of time $t$. Then, the asymptotic stability of the system with respect to $s$ is proven by Theorem 1 [Xi 1993]. This asymptotic stability is proven regardless of the Internet random delay [Xi and Tarn 1999]. □

On the other hand, since the event-based control approach does not provide a way to predict future system responses, this thesis introduces a design criterion. This design criterion defines the internal off-line frequency and sampling period $S_{sim}$ of the Simulation block, and it is formally stated with the following theorem:

**Theorem 2:** If the event-based system is asymptotically stable with an action reference $s$ and the Simulation block starts predicting from event $s_n$ ($n \in \mathbb{Z}^+$) the reconstructed effect of future events, then the simulated system must output a simulated effect $x'(t)$ on the simulated remote environment, which is band-limited by

$$w'_\text{max} \geq w_{\text{max}}$$

and the internal off-line sampling period $S_{sim}$ of the Simulation block must satisfy the following sampling condition:

$$\frac{\pi}{KS_{sim}} \geq \frac{\pi}{KS}$$

**Proof:**

**Necessary:** If the real event-based system is action-referenced to $s$ with a sampling period $S$, assume that the deviations of the non-uniform samples of the sensed effect $x(t)$ (produced by the real system) from uniform samples are $\delta_n$ [Elhajj et al. 2003, Elhajj 2002]. In addition, assume that the sampling period $S$ defines the function [Unser 2000, Corporation 2005]:

$$\beta(s) = \sum_{n=-\infty}^{\infty} \delta_n \frac{\sin w_{\text{max}}(s-nKS)}{w_{\text{max}}(s-nKS)},$$

where sample values of $\beta(s)$ are $\delta_n$,

$$\beta(nKS) = \delta_n$$

and the transformation between the $s$-axis and $t$-axis is

$$t = s - \beta(s).$$

It follows that

if $s = nKS$, then $t = nKS - \beta(nKS) = nKS - \delta_n$.

Therefore, Eq. 3.18 transforms the point $nKS$ into the point $nKS - \delta_n$ of the $x$-axis [Elhajj 2002].
Since $s_n < s_{n+1} \Rightarrow t_n < t_{n+1}$ and two events cannot occur at the same time instant, Eq. 3.18 is an increasing function of $s$ and its inverse Fourier transform exists:

$$s = \gamma(t).$$

(3.20)

Define the function

$$g(s) \equiv x[s - \beta(s)] = x(t),$$

that implies

$$g(nKS) = x(nKS - \delta_n).$$

(3.22)

Hence, $g(s)$ is known at a sequence of equidistant points $s = nKS$. If this signal is band-limited by $w_{\text{max}}$, then $S$ can be chosen such that the sampling theorem condition is satisfied [Unser 2000, Corporation 2005]:

$$\frac{\pi}{KS} \geq w_{\text{max}},$$

(3.23)

then the following reconstruction equation (the cardinal series expansion) will hold:

$$g(s) = \sum_{n=-\infty}^{\infty} x(nKS - \delta_n) \sin \frac{w_{\text{max}}(s - nKS)}{w_{\text{max}}(s - nKS)}.$$  

(3.24)

Since Eq. 3.15 implies that $nKSS_{\text{sim}} \leq nKS$, then Eq. 3.23 and Eq. 3.24 will hold for a simulated reconstructed signal $g'(s)$ ($y_p(s)$ in the architecture scheme) defined by $nKSS_{\text{sim}} \leq nKS$, as follows.

Suppose that the simulated effect $x'(t)$ is band-limited by $w'_{\text{max}} = w_{\text{max}}$. If

$$\frac{\pi}{KSS_{\text{sim}}} = \frac{\pi}{KS},$$

(3.25)

then the simulated system starting from $s_n$ will predict the effect of future event $s'_{n+1}$. This means that the simulated reconstructed signal $g'(s)$ has reached event $s'_{n+1}$, as shown in Figure 3.16. This implies that the simulated output $x'(t)$ had the necessary effect to reach a predefined state of the simulated robotic system (distance, position, among others).

Now, suppose that the simulated effect $x'(t)$ is band-limited by $w'_{\text{max}} > w_{\text{max}}$. If

$$\frac{\pi}{KSS_{\text{sim}}} > \frac{\pi}{KS},$$

(3.26)

then the simulated system will predict the reconstructed effects of events $s'_{n+1}, s'_{n+2}, ..., s'_{j}$ in a period $S$ if

$$S_{\text{sim}} = \frac{1}{j}S \text{ and } w'_{\text{max}} = jw_{\text{max}},$$

(3.27)

as shown in Figure 3.16.
Sufficient: If the real system is action referenced to $s$ with a sampling period $S$, suppose that $w'_{\text{max}} = w_{\text{max}}$ and

$$\frac{\pi}{KS_{\text{sim}}} < \frac{\pi}{KS},$$

then Eq. 3.23 and Eq. 3.24 will not hold with $nKS_{\text{sim}} > nKS$, implying that the simulated system will provide an aliased reconstructed effect $g'(s)$ of the real reconstructed effect $g(s)$, as shown in Figure 3.17. Even if $w'_{\text{max}} > w_{\text{max}}$, the simulation will reconstruct an aliased effect $g'(s)$ of the real system reconstructed effect $g(s)$.

As a consequence, the simulated system has to be designed with an internal iteration frequency $w'_{\text{max}}$ greater or equal to the maximum frequency $w_{\text{max}}$ and with a internal sampling period less or equal to the sampling period of the action reference $s$ of the real event-based system. □

Thus, the proposed teleoperation architecture can work in the following operation modes:

**Bilateral Teleoperation:** Operator and Remote Environment close a control loop, where the sampling period $S$ of the non-time based reference $s$ must be chosen, based on the sampling condition of Eq. 3.23 [Unser 2000, Elhajj 2002].

**Teleoperation + Simulation:** A main control loop is closed by Operator and Remote Environment and an off-line simulated control loop is closed in the Simulation procedure in the Operator (Guest) system. The sampling condition of the main control is chosen using Eq. 3.23, while the internal iteration frequency and simulated sampling period $nKS_{\text{sim}}$ of the Simulation block must be chosen, based on the following condition:

$$w'_{\text{max}} \geq w_{\text{max}} \Rightarrow \frac{\pi}{KS_{\text{sim}}} \geq \frac{\pi}{KS}$$

**Supervisory Control:** Sensory Conversion, Output Device, Operator, Input Device, Input Processing, and Simulation blocks of the Operator (Guest) side work as a planner as shown in Figure 3.13, and the overall closed-loop system is event-based referenced to $s$, based on the sampling condition of Eq. 3.23.

### 3.6 VL Modeling Conclusion

This chapter has provided a complete definition of VLs for telerobotics and a control basis for their development. The generic and modular VL model was translated into UML diagrams. The model was also related to event-based planning and control theory for Internet-based telerobotics through a general event-based teleoperation architecture.
The model provides a panoramic view that describes processes and devices in a VL system. The model translation into standard UML diagrams introduces a formal definition of every aspect of VL systems. The class diagram provided an object-oriented overview of procedures and devices as components of the VL class. The collaboration diagrams have shown interactions between procedures and devices composing VL. These interactions were displayed as conceptual associations between class instantiations. The deployment diagram showed distribution of software and hardware. Use case and sequence diagrams relate VL development with the True Experiment concept and showed multiple interactions between VL subsystems and the user. State changes of VL components are described by UML statechart diagram.

Control design foundations were also introduced using a general teleoperation architecture to generate design procedures. This architecture was defined by taking into account VL model components and the event-based control approach. The event-based control provided conditions to implement stable and synchronized VL systems, and the necessary compatibility with the Petri Net modeling and analysis technique.

These different modeling techniques (UML and Petri Net) are used to design VL applications by a development methodology that will be introduced in Chapter 4. The proposed methodology will use the VL framework as reference and will customize it to fulfill experiment specifications. The methodology will take advantage of Petri Net analysis and control techniques to systematize the dynamic design and implementation of VLs for telerobotic issues.

Furthermore, the proposed methodology will be independent of any standard inter-communication technologies (e.g. CORBA) and, at the same time, will be expandable and applicable to design VL applications based on those technologies because of its object-oriented approach.
Figure 3.16: Comparison between real and simulated reconstructed output signals. The first row describes the real output effect \( x(t) \) referenced to \( t \). Second row represents a simulated signal \( g'(s) \) referenced to \( s \), reconstructed with the same sampling period than the real reconstructed signal \( g(s) \). Third row describes a simulated output effect \( x'(t) \) band limited by \( w_{\text{max}}' > w_{\text{max}} \). Fourth row describes a simulated signal \( g'(s) \) reconstructed with a sampling period minor than \( g(s) \).
Figure 3.17: Comparison between real and simulated reconstructed output signals. The first row describes the real output effect $x(t)$ referenced to $t$. Second row represents the effect $g(s)$ referenced to $s$, reconstructed with a sampling period $S$. Third row describes an aliased simulated signal $g'(s)$ reconstructed with a sampling period greater than $g(s)$. 
Chapter 4
Development Methodology

Challenges faced by developers when designing and building VLs for Internet-based telerobotic experimentation are quite diverse and more complicated than the problems in design that show up in the development of traditional teleoperation applications. From the beginning, VL development must take into account main functionalities needed to perform a true experimentation: mount, define, and execute. These functionalities embody complex processes that are necessary to remotely control and monitor the experiment, run activities to set and modify experiment settings, and request and gather information regarding experiment behavior. In addition, these functionalities must be integrated into a complex system (the VL) that allows performing them in a sequential and synchronized way.

To reduce and guide efforts required for designing and implementing new VL applications, this chapter introduces a development methodology based on the proposed generic VL framework for prioritizing the sequential execution of experiment functionalities into VL designs. The proposed methodology was described in [Borstel and Gordillo 2004b] and consists of four phases for modeling, analyzing, and designing VL applications, as shown in Figure 4.1.

In the first phase, the experiment is conceptually designed to accomplish VL functionalities and is analyzed to define processes and methods needed to perform experiment objectives. This phase generates the experiment process flow chart, several specification tables, and hierarchical charts. These tables and hierarchical charts include functional descriptions and specifications of necessary processes and devices per functionality.

In the second phase, the methodology transforms the process flow chart, specification tables, and hierarchical charts into the proposed UML framework to provide a standard representation of the VL design. VL main functionalities (mount, define, and execute) are treated as subsystems, which are represented by customized instantiations of the VL framework. In this phase, the inherent procedures and devices of these subsystems are identified, based on their functional descriptions as class
Figure 4.1: Methodology phases for developing VLs for telerobotic experimentation over the Internet. The method consist of four phases to develop VLs from their abstract design to their implementation.

- **Phase 1**: Experiment Design and Functionality Analysis
  - Experiment Design
  - Processes | Functions
  - Define
  - Execute

- **Phase 2**: Experiment Subsystems Identification into VL Framework
  - UML Framework
    - Classes: Collaboration, Deployment
    - Statechart | Sequence

- **Phase 3**: Conversion into Dynamic Model and Analysis
  - Petri Net (Mount)
  - Petri Net (Define)
  - Petri Net (Execute)

- **Phase 4**: Composition, Analysis and Control of Dynamic Model
  - Composed Petri Net
  - Controller Net
  - VL Design

**VL Complete Design**

**Structural VL Design (UML)** + **Behavioral VL Design (Petri Net)**

The third phase extracts the detailed behavior of each subsystem, represented by UML statechart diagrams, to convert it into Petri Net formalism. This formalism provides tools to quantitatively and qualitatively analyze the dynamics of each VL subsystem design. This step is necessary because of the lack of analytical techniques in UML notation. Furthermore, UML statechart diagrams are converted into Petri Net elements to provide a way to analyze and validate required behavioral properties for event-based Internet-based systems.

A merge of VL subsystem Petri Nets and an analysis of the resulting composed Petri Net are made in the last phase. Petri Net techniques, synchronous transition composition, and parallel place fusion, are used to integrate VL subsystems into a composed Petri Net. This composed net is analyzed to check that VL
subsystems will be performed sequentially and correctly. VL subsystems are DES (Discrete Event Systems) with discrete state spaces and event-driven state transitions [Moody and Ansalklis 1998]. If VL subsystems have not required behavioral properties to accomplish experiment goals, then the developer can apply supervisory control methods for DES to synthesize controller nets, which provide the necessary control to execute them appropriately. Thus, the controlled composed Petri Net is a tested VL dynamic design.

Once developmental phases are finished, the VL is defined statically and dynamically by UML and Petri Net formalisms. VL subsystems are defined by customized UML frameworks, which specify software structure and functionality to provide a guide for their implementation into object-oriented programming languages. Generated Petri Nets specify the dynamics of the VL and allow their analysis and verification before their implementation. Petri Net formalism is also compatible with object-oriented programming languages and their event control procedures. Therefore, the composed Petri Net can be directly used by developers to implement the necessary control among VL procedures.

4.1 Experiment Design and Functionality Analysis

The proposed methodology starts with the conceptual design and analysis of the telerobotic experiment. This conceptual design must take into account experiment functionalities to achieve a true experiment [Hicks 1982], which were specified by the UML use case diagram of the VL framework. A general process flow chart, functional descriptions, and computer-based techniques among other hardware and software specifications are then described. This design is analyzed to separate Mount, Define and Execute functionalities and treat them independently. In the experiment analysis, computer-based techniques and devices are treated as processes, which have specific functions in the VL application. These processes are described by hierarchical charts, which show main functionalities at the top, and procedures with their specific functions at the bottom.

Once the process flow chart is finished, experiment functionalities are described by tables that include their procedures and specifications. These functionalities are also described by hierarchical charts, which specify relationships among procedures and their specific functions. The methodology continues to translate this design into a formal standard modeling language.

4.2 Subsystem Identification in the VL Framework

The second phase in development methodology identifies objects in the experiment and relates them to the proposed UML framework to generate a formal VL model.
Chapter 4. Development Methodology

Figure 4.2: Diagrams of the first developmental phase: a) process flow charts, b) tables filled with the functionality specifications, c) hierarchical charts of experiment functionalities.

Generated diagrams and charts, per each VL functionality, are considered as subsystems. They are related to the VL framework to create customized instantiations. Three modeling stages are specified in this phase: structure, dynamics, and operation modeling.

At the first modeling stage, processes from hierarchical charts are identified as class instantiations of the VL framework, and their specific functions are treated as operation methods of VL classes. Then, the VL class domain is created to describe the static structure of VL subsystems. Specifications recovered from the functionality tables are treated as attributes of identified VL classes and operation methods to generate customized class diagrams of the VL framework. In these class diagrams, interrelationships among processes described by the experiment process flow chart are interpreted as class associations, as shown in Figure 4.3.

In the second modeling stage, information contained in the experiment description is used to construct UML sequence diagrams. If complex behaviors that cannot be represented by standard UML sequence diagrams exist, e.g., choice, then extensions made by [Jeng and Lu 2002] should be used. Thus, these sequence diagrams will formally represent the desired behavior of the VL design and will provide foundations to create collaboration diagrams, which represent message flow among class instantia-
4.3 Conversion into Dynamic Model and Analysis

Although UML is convenient for modeling complex systems and is widely accepted by the industrial sector, it is not equipped with necessary techniques to qualitatively and quantitatively analyze behavioral properties of designed systems. Therefore, UML statechart diagrams are converted into Petri Nets to take advantage of their powerful analysis and design techniques.
Chapter 4. Development Methodology

Figure 4.4: Second modeling stage in the subsystem identification in the VL framework. Informal diagrams are translated into UML collaboration and sequence diagrams for dynamic modeling. This procedure is performed for each experiment functionality.

Informal and Formal Definition:
- Experiment description
- Flow chart
- Specification tables
- Hierarchical charts
- Class diagram
- Sequence diagram
- Collaboration diagram
- Deployment diagram

Figure 4.5: Third modeling stage in the subsystem identification phase. The informal and formal experiment definition is used to generate detailed operation models (statechart diagram) for each functionality.
Petri Nets are powerful graphical and mathematical modeling tools to describe and analyze discrete systems, such as DES. They are able to visually show structure and dynamics of many systems, such as computer-based systems, and make it possible to generate state equations and mathematical models to represent and specify their behavior.

Since this work proposes a complete development of VL applications, it translates UML statechart diagrams into the Petri Net formalism to cover every aspect of VL dynamics design. The following subsections formally introduce the proposed transformation, based on state machine theory foundations.

### 4.3.1 Statechart diagram-Petri Net Conversion

Petri Nets are classified according to structural properties involving the ways arcs connect places and transitions. A Petri Net represents a state machine if all transitions have one input and one output. Significant nodes in a state machine are places. Each transition allows tokens to flow from one place to another, but a token in a particular place may enable multiple transitions. This is referred to as a conflict, decision, or choice, depending on applications. All finite automata can be described as Petri Net state machines [Murata 1989].

Several methods for converting UML diagrams into Petri Nets have been proposed. Some of them convert UML activity, use case, class, deployment, and statechart diagrams into multiple subclasses of Petri Nets (timed, labeled, colored) [Boccalatte et al. 1999, Bondavalli et al. 1999, Bordbar et al. 2000, Chang et al. 2000, Lu 2001].

The proposed transformation method converts UML statechart diagrams (behaving as state machines) into low level and unmodified Petri Nets to translate system dynamics. These low level Petri Nets are preferred because their multiple analysis techniques are supported. More analytical techniques have been developed for low level Petri nets than high level and modified nets, such as timed, labeled, and colored.

Several conversion rules for statechart diagrams and extensions for sequence diagrams that allow representing concurrency, synchronization, non-determinism, and other properties of Petri Nets were introduced by the work of [Jeng and Lu 2002] work. However, these rules were not formally established. This study will propose a formal conversion to generate ordinary Petri Nets from statechart diagrams behaving as state machines, meaning that all transitions have one input state and one output state. Therefore, statechart diagrams are defined by initial and final pseudostates, transitions, and simple states without extensions, such as state orthogonality for representing concurrency [Booch et al. 1999].

Since UML statecharts and Petri Nets are based on foundations of finite state
machines and contain states and transitions, the proposed UML statechart-Petri Net translation is simple. Each state is converted into a Petri Net place and each transition into a Petri Net transition.

![Statechart-Petri Net Conversion](image)

Figure 4.6: UML statechart-Petri Net conversion. The formal conversion translates UML statecharts behaving as machine states into ordinary Petri Nets. System states are converted into places and transitions are converted into Petri Net transitions.

**Definition 2:** Formally, the proposed conversion function is denoted as $G(ST) = PN$, the statechart diagram (state machine) is $ST = (S, TST, A, i, f)$, which is converted into a $PN = (P, TPN, \xi, M_o, \alpha)$, where:

$$P = \{s | s \in S\} \quad (4.1)$$

$$TPN = \{t | t \in TST\} \quad (4.2)$$

$$\alpha = \{x | x \in \Sigma ST\} \quad (4.3)$$

$$\xi = \{a | a \in \{(s, t) \text{ is defined}\} \cup \{(t, s) \text{ is defined}\} \text{ and } s \in S \text{ and } t \in TST\} \quad (4.4)$$

$$\mu(p) = \begin{cases} 1 & \text{if } p = i \text{ and } i \in S \\ 0 & \text{otherwise} \end{cases} \quad (4.5)$$

In automata theory, two automata are said to be *equivalent* if they generate and accept the same languages [Cassandras and Lafortune 2001]. Formally, Automata $A_1$ and $A_2$ are said to be equivalent if

$$L(A_1) = L(A_2). \quad (4.6)$$

Therefore, the equivalence between the Petri Net $PT$ generated by $G(ST)$ and the original statechart $ST$ can be formally stated by the following theorem.
Theorem 3: If statechart diagram ST, behaving as state machine automaton, is represented by transitions with one input and one output states and states that include the start and final states, then ST is equivalent to an ordinary Petri Net PN generated by $G(ST)$, when

$$L(ST) = L(PN). \quad (4.7)$$

Proof:

This is a deductive proof based on set theory. For two sets $S_1$ and $S_2$, if $S_1 \subseteq S_2$ and $S_1 \supseteq S_2$, then $S_1 = S_2$. This proof will demonstrate that if $L(ST) \subseteq L(PN)$ and $L(ST) \supseteq L(PN)$, then $L(ST) = L(PN)$.

- To prove $L(ST) \subseteq L(PN)$, let $x \in L(ST)$; then this first part will prove that $x \in L(PN)$. By language definition of a statechart diagram: if $x \in L(ST)$, then $x \in \Sigma^*_{ST}$ and $\exists y \in S$, such that $\Gamma(y) = x$ and $\hat{f}(i, x)$ is defined. In the same way, function $G(ST)$ defines that $T_{PN} = T_{ST}$, which implies that $T_{PN} = T_{ST}$. Furthermore, $G(ST)$ defines $\alpha = \Sigma_{ST}$ and since the labeling function $\alpha$ is extended to $\alpha = T_{PN}^* \rightarrow \Sigma_{PN}$, this implies that the extended function $\alpha = \Sigma_{PN} = \Sigma_{ST}$; thus $x \in \Sigma_{PN}^*$ and $\exists \tau \in T_{PN}^*$, such that $\alpha(\tau) = x$. Finally, since $\hat{f}(i, x)$ is defined and the conversion function defines $G : i \rightarrow M_o$, it implies that $M_o[i \tau]$ is also defined. Therefore, $x \in L(PN)$, which implies that $L(ST) \subseteq L(PN)$, when $G(ST) = PN$.

- To prove $L(ST) \supseteq L(PN)$, let $x \in L(PN)$; then this second part will prove that $x \in L(ST)$. By language definition of a Petri Net: if $x \in L(PN)$, then $x \in \Sigma_{PN}$ and $\exists \tau \in T_{PN}^*$, such that $\alpha(\tau) = x$ and $M_o[\tau]$ is defined. In the same way, function $G(ST)$ produces $T_{ST} = T_{PN}$, which implies that $T_{ST} = T_{PN}^*$. Furthermore, $G(ST)$ defines $\alpha = \Sigma_{ST}$ and, since the labeling function $\alpha$ is extended to $\alpha = T_{PN}^* \rightarrow \Sigma_{PN}$, this implies that the extended function $\alpha = \Sigma_{PN} = \Sigma_{ST}$; thus $x \in \Sigma_{ST}^*$. Finally, since $M_o[\tau]$ is defined and the conversion function defines $G : M_o \rightarrow i$, it implies that $\hat{f}(i, x)$ is also defined and that $\exists y \in S$, such that $\Gamma(y) = x$. Therefore, $x \in L(ST)$, which implies that $L(PN) \supseteq L(ST)$, when $G(ST) = PN$.

Therefore, if $G(ST) = PN \Rightarrow L(ST) = L(PN)$.  

This result implies that a state machine automaton represented by a UML statechart diagram is equivalent to an ordinary Petri Net generated by the proposed conversion method. The VL statechart diagram shown in Figure 3.12 was translated into a Petri Net graph using the proposed method, as shown in Figure 4.7.

Since this conversion results in a Petri Net for each VL subsystem represented by a statechart diagram, these Petri Nets are convenient for further qualitative and quantitative analysis as event-based VL subsystem designs.
Figure 4.7: Petri Net generated from UML statechart diagram of Figure 3.12, using the statechart-Petri Net conversion method.
4.3.2 Petri Net Analysis

Generated Petri Nets are used to make a dynamic analysis of VL subsystems to verify their behavioral properties, such as liveness, and boundedness, among others. Petri Nets support analysis of many important properties associated with the systems that they model. These properties are grouped into those that are dependent on initial marking (behavioral properties) and those that are independent of initial marking (structural properties). The following behavioral properties are important for VL design:

**Boundedness:** a Petri Net $PN$ is said to be $k$-bounded or simply-bounded if the number of tokens do not exceed a finite number $k$ for any reachable marking $M$ from the initial marking $M_0$. This implies that there is no accumulation of tokens in places at any time. A Petri Net with 1-bounded property is said to be “safe”.

**Liveness:** the concept is closely related to a complete absence of deadlocks in Operating Systems (OS). A Petri Net $PN$ is said to be live if, no matter what marking $M$ has been reached from the initial marking $M_0$, it is possible to ultimately fire any transition $t \in T_{PN}$ by progressing through some further firing sequence $\tau \in T_{PN}$. This means that a live Petri Net guarantees deadlock-free operation, no matter what firing sequence is chosen. Relaxed liveness conditions were presented in [Murata 1989], where different levels of liveness were defined, as follows. A transition $t$ in a Petri Net is said to be:

1. **dead** ($L0$-live) if $t$ can never be fired in any transition firing sequence of the Petri Net.
2. **L1-live** (potentially fireable) if $t$ can be fired at least once in some firing sequence.
3. **L2-live** if, given any positive integer $k$, $t$ can be fired at least $k$ times in some firing sequence.
4. **L3-live** if $t$ appears infinitely, often in some firing sequence.
5. **L4-live** or **live** if $t$ is L1-live for every marking of the Petri Net.

Analysis methods for Petri Nets may be classified into the following three groups: the coverability (reachability) tree method, the matrix-equation approach, and reduction of decomposition techniques [Murata 1989]. The first method involves the enumeration of all reachable markings or their coverable markings. This method is simple and it is capable of describing significant behavioral properties in a Petri Net.

The coverability tree method is based on the marking $M$ of Petri Nets. Given a Petri Net $PN$, from the initial marking $M_0$, as many new markings can be obtained as the number of enabled transitions. From each new marking, more markings can be reached. This process results in a tree representation of the markings. Nodes represent markings generated from $M_0$ as the root and its successors, and each arc represents a
transition firing, which transforms one marking into another, as shown in Figure 4.8.

The coverability tree is closely related to $L1$ – liveness (potential firability). Let $M$ be the minimum marking needed to enable transition $t$. Then $t$ is dead (not $L1$ – live) if, and only if, $M$ is not coverable.

Figure 4.8: An ordinary Petri Net and its coverability tree. The Petri Net is safe (1-bounded) and not live.

4.3.3 Event-Based VL Subsystem Analysis Using Petri Nets

A coverability tree analysis determines if the system is event-synchronized. The event-synchronized property [Elhajj et al. 2002a] is necessary to establish that control and feedback signals are synchronized at the same event-based reference at any instant. This is translated into two requirements: first, no two instantiations of the same signal can coexist anywhere in the network, and second, every feedback instant or sample should generate one and only one command response and vice versa. These requirements can be translated into Petri Nets properties [Elhajj 2002]:

- There is a need to ensure that the Petri Net will not dead-lock, which formally means the net has to be shown as live.

- It is required not to have two instantiations of the same signal at any point; this means no accumulation of signals at any point within the system. This translates into having either one or no tokens in any place in the net. For this to be true, the Petri Net has to be safe.

However, the event-based control analysis for VL subsystems has different requirements. Since each subsystem composing a VL system has a goal to define one or several independent variables of the experiment, they cannot be represented by live Petri Nets, e.g., VL subsystems will not always return to their first marking $M_0$. Therefore, their analysis must consider a different level of liveness.

The event-synchronization property of an event-based VL system represented by a Petri Net is formally stated with the following theorem.
Theorem 4: A VL subsystem is event-synchronous if and only if the Petri Net describing the system data flow is safe, it represents a state machine automaton, and has a firable transition sequence $\tau$ that allows $M_0[\tau > M$, where $M$ is the marking to reach the VL subsystem goal (final state).

Proof:

Necessary. Assume a Petri Net is not safe; this implies that there exists at least one place that has more than one token at the same time, and since tokens represent signals in the system, there are two instances of the same signal at the same time in the system having different event-based references. Therefore, the system is not event-synchronous.

Sufficient. If a Petri Net is safe; it implies that the Petri Net is 1-bounded, which means that the number of tokens in each place does not exceed 1 for any reachable marking from the initial marking $M_0$. For the data flow, this implies that there can only be one instance of a signal at any point in the VL subsystem. In addition, if a firable transition sequence $\tau$ that allows $M_0[\tau > M$ exists, which means that there is a path from the initial marking $M_0$ to the goal marking $M$, it implies that the goal state is reachable. Functionality of the VL subsystem implies that the signal (token) has fired the necessary transitions (procedures) to reach the VL subsystem goal. However, this does not guarantee that other instances of a signal do not exist at other places. This is accomplished by requiring the Petri Net to be a state machine automaton. A state machine is a Petri Net in which all transitions have one input and one output place, implying that a transition (procedure) cannot generate more than one signal (token). So a new instance of a signal will not be generated unless the earlier one has been processed. Therefore, this ensures the second requirement for event-synchronization.

The L1-live definition (potential firability) is considered for transition $t$ that is fired to reach the goal state of the VL subsystem. This property is deduced using the coverability tree analysis method.

### 4.4 Composition, Analysis and Control of a Dynamic Model

This methodology phase uses synchronous transition composition and parallel place fusion procedures of Petri Nets to join VL subsystems. This allows performing true experimentation over the Internet. This composition integrates VL subsystems in a way that transitions with the same label (meaning the same operation method) can be merged into one transition; places can be merged similarly.

The composed Petri Net that results from the composition is analyzed to verify
previously described behavioral properties, which are important for maintaining stability and synchronization and avoiding deadlocks in VL applications.

Results from this analysis, e.g., coverability tree method, provide a way to deduce forbidden states of the composed Petri Net. These forbidden states are system conditions, which are incompatible with experiment specifications. They must be avoided to accomplish the experiment goal. These forbidden states are represented as system constraints in an if-then notation and used to design controller nets, which are incorporated into the composed net to prevent them. The composed and controlled Petri Net is analyzed to verify the correct execution of each VL subsystem.

![Diagram](image)

Figure 4.9: Composition, analysis, and control stages in the fourth phase of the development methodology.

### 4.4.1 Synchronous Transition Composition Definition

VL subsystems for mounting, defining, and executing are merged, using Petri Net design procedures; if VL subsystems share similar transitions and places, then these transitions and places are merged, using synchronous transition composition and place fusion to create a VL system able to provide necessary functionalities to perform experiments remotely. Notice that, if VL subsystems do not share any transition or place, they can be merged by adding places generated by the design method discussed in [Elhajj 2002, Elhajj et al. 2003]. The formal description of the synchronous transition composition is given by the following definition [Kumar and Holloway 1996].

Synchronous transition composition of Petri Nets $PN_1 = (P_1, T_1, \xi_1, M_0, \alpha_1)$ and $PN_2 = (P_2, T_2, \xi_2, M_0, \alpha_2)$ is another Petri Net $PN_1 || PN_2 = PN = (P, T, \xi, M_0, \alpha)$, where:

\[
P = P_1 \cup P_2
\]

\[
T = \{(t_1, t_2) \in T_1 \times T_2 | \alpha_1(t_1) = \alpha_2(t_2)\} \text{ and } \alpha((t_1, t_2)) = \alpha_1(t_1) = \alpha_2(t_2)
\]

\[
\xi = \{(p, (t_1, t_2)) \in P \times T | (p, t_1) \in \xi_1 \text{ or } (p, t_2) \in \xi_2\} \cup \{(t_1, t_2), p) \in T \times P | (t_1, p) \in \xi_1 \text{ or } (t_2, p) \in \xi_2\}
\]

\[
\mu(p) = \begin{cases} 
\mu_1(p) & \text{if } p \in P_1 \\
\mu_2(p) & \text{if } p \in P_2
\end{cases}
\]

The set of places in the synchronized net equals the union of those in the individual nets. The synchronized net replaces a pair of transitions with the same label, but
in separate nets, with a single transition in the new net. This single transition has the “input” places and “output” places as a respective union of those from the previous transition pair. Note that several transitions may exist in each net with the same label, in which case, one transition exists in the synchronized net for each transition pair combination. It is straightforward to show that the language of the synchronized net \( PN \) satisfies 
\[
L(PN) = L(PN_1) \cap L(PN_2) \quad \text{[Peterson 1981].}
\]
Also, \( PN \) is deterministic whenever \( PN_1 \) and \( PN_2 \) are deterministic [Kumar and Holloway 1996].

In this work, the initial marking of \( PN_2 \) is not used to define the initial marking of the composed Petri Net. The VL subsystems must be executed sequentially: first, the VL subsystem to mount the experiment, then the experiment definition subsystem, and lastly the VL subsystem to execute the experiment. Therefore, the first subsystem defines the initial state of the entire VL system; this implies that for every Petri Net synchronously composed, the first net \( (PN_1) \) inherits its initial marking to the composed net. Therefore, Eq. 4.11 is modified and the initial marking for the synchronous composition of VL subsystems is defined by:

\[
\mu(p) = \begin{cases} 
\mu_1(p) & \text{if } p \in P_1 \\
0 & \text{if } p \in P_2
\end{cases}
\]

(4.12)

Figure 4.10: The original transition synchronous composition of two nets \( PN_1 \) (a) and \( PN_2 \) (b) with similarly labeled transition function \( \alpha_1 = \alpha_2 \). The wide arrow indicates a Petri Net \( PN \) resulting from transition synchronous composition \( PN_1 \| PN_2 = PN \) with \( \alpha = \alpha_1 = \alpha_2 \).

4.4.2 Parallel Place Fusion Definition

If \( p_a \in PN_1 \) and \( p_b \in PN_2 \) are places that, once the synchronous composition \( PN_1 \| PN_2 \) is performed, share the same input transition \( p_a = p_b \) and the same output transition \( p_a^o = p_b^o \), then these places can be reduced into one place based on the parallel place fusion procedure described in [Murata 1989], as shown in Figure
4.11. Since VL subsystem Petri Nets have places representing the same system states, if places $p_a$ and $p_b$ represent the same system state, then the fused place represents the same state.

![Diagram showing parallel place fusion procedure](image)

Figure 4.11: Parallel place fusion procedure; two places $(p_a, p_b)$ having the same output transition and the same input transition, are fused into one place $p_{ab}$. If both places represent the same system state, then the fused place represents that state.

**Definition 3:** Formally, the modified synchronous composition applied on two Petri Nets $PN_1$ and $PN_2$ and the parallel place fusion procedure applied to the resulting net $PN_3$ are denoted as $PN_1 \parallel^+ PN_2 = PN_3$, where $PN_3$ is the net result of the synchronous transition composition and the parallel place fusion procedure applied on itself.

The modified synchronous composition and place fusion procedures generate a new composed $PN_3$ with a new marking $M$. This composed Petri Net represents a complete VL system. In practice, two rules are recommended to establish the new marking $M$ of the composed Petri Net $PN_3$:

1. Initial marking ($M_o$) of $PN_3$ must be referenced to the first merged Petri Net $PN_1$. A token is always placed where $PN_1$ starts (the place $p \in P_1$ with $\mu(p_0) = 1$ in $M_{o1}$).

2. Marking $M$ must be referenced to the first merged Petri Net $PN_1$. The numeric labeling must begin at place $p_0$ where $PN_1$ starts. Then, further numeric labels must be reorganized to create a joined marking $M$ of $PN_3$.

### 4.4.3 Behavioral Analysis of Composed Petri Net

Synchronous composition and place fusion do not guarantee that VL subsystems will be executed in an appropriate way. Therefore, it is necessary to analyze and design additional nets to control the composed Petri Net.
4.4. Composition, Analysis and Control of a Dynamic Model

The composed Petri Net is analyzed to verify that VL subsystems perform their functionalities (mount, define, and execute) correctly. This methodology phase essentially verifies that VL subsystems are performed sequentially in a synchronized way. If these properties do not satisfy experiment specifications, then supervisory control techniques for DES are applied to satisfy specifications. Therefore, the methodology goes back to the analysis stage previously described to verify Petri Net behavioral properties. This analysis is done until the composed net satisfies experiment specifications generating a final event-based VL design.

The analysis performed using the coverability tree method provides a list of specific forbidden constraints for the VL system that are converted into linear equations used to synthesize controller structures.

4.4.4 Control Design for Composed Petri Net

Because of the synchronous composition, some transitions in the composed Petri Net may fire in a non-deterministic way, which is not convenient for experiment specifications. Therefore, it is necessary to implement additional structures to control transitions with non-deterministic behavior. These structures are based on supervisory control theory for a Discrete Event System (DES) to provide a systematic control design on the sequential execution of VL subsystems.

This stage provides control design on the composed Petri Net; it allows the execution of each VL subsystem in a sequential and synchronized way to obtain an appropriate behavior when the experiment is performed.

4.4.5 Supervisory Control of DES using Petri Nets

A DES is a system whose state space is discrete and whose state can only change as a result of asynchronous, instantaneously occurring events over time. A DES satisfies the following two properties: its state space is a discrete set and the state transition is event-driven. Sample paths of DES are typically piecewise constant functions of time. Conventional differential equations are not suitable for describing such discontinuous behavior [Cassandras and Lafortune 2001]. Examples of a DES are computer, communication, manufacturing, software, and traffic systems.

A VL system is a computer-based complex system behaving as a DES. UML has the ability to represent VL dynamics using state machine automata (statechart diagram). However, a powerful alternative to a statechart diagram for models of a DES is provided by Petri Nets. Like an automaton, a Petri Net is a device that manipulates events according to certain rules. The dynamic behavior of a Petri Net is described by using tokens in places to enable transitions, which then cause tokens to move around.
The control theory for a DES to design *supervisors* (controller nets) that prevent *forbidden* or undesirable states from occurring in an automaton-modeled system is called *Supervisory Control* of the DES. It is possible to construct an automaton supervisor or controller net using several synthesizing methods, such as the ones described in [Moody and Ansalklis 1998]. The controller net modifies the system behavior via a feedback loop, as shown in Figure 4.12. The controller net consists of places that are connected to the transitions of a system, insuring that the state vector of the system remains within bounds established by a set of linear constraints. Formally, the supervisory control makes the language generated by the DES automaton $PN$ be restricted to a subset of $L(PN)$ by the controller net $CN$.

Typical supervisory control procedures for a DES construct *maximally permissible* controller nets; this means that maximally permissible controllers must not affect the system behavior, except when forbidden states are presented, as they are defined by constraints. Because of the increment of non-determinism in the composed Petri Net produced by the synchronous composition, it is necessary to synthesize controller nets to control the firability of transitions that do not behave according to experiment specifications and that may lead to deadlocks in the VL system. Therefore, *non-maximally permissible* controller nets are needed to modify the system behavior by enabling or disabling specific segments of the composed Petri Net to meet experiment specifications [Moody and Ansalklis 1998]. A logical control method that synthesizes a non-maximally permissible controller net is proposed. This proposed control method will be explained in a further subsection.

![Figure 4.12: Controller net monitoring and controlling the composed Petri Net.](image)

### 4.4.6 Constraint Forms to Synthesize Controller Nets

Supervisory control methods synthesize a controlled net to avoid forbidden states of the designed system. These forbidden states are described as control constraints that the system must satisfy. Control constraints are the logical conjunction of separated linear inequality constraints. For example, consider the supervisory control goal of
restricting the reachable marking (number of tokens) in place \( p \) of system \( PN \), such that:

\[
\mu(p) \leq b,
\]

where \( b \in \mathbb{Z}^+ \) and \( \mu(p) \in \mathbb{Z}^+ \). Eq. 4.13 represents a constraint where the marking of place \( p \) is less than or equal to the constant \( b \). This constraint can be transformed into an equality by introducing a non-negative slack variable into it, such as the marking of a new place \( p_c \) called controller or monitor place. Then, the constraint becomes

\[
\mu(p) + \mu(p_c) = b
\]

and place \( p_c \) holds the extra tokens required to meet the equality. The rules of Petri Nets state evolution insure that \( \mu(p_c) \) is non-negative by definition. Therefore, if equality in Eq. 4.14 can be forced on the marking of the controlled system, the sum of tokens in places \( p \) and \( p_c \) will always be less than or equal to \( b \).

On the other hand, there are constraint forms that involve the firing vector of Petri Nets as well as or opposed to the places, e.g., the constraint form

\[
\mu(p) + t \leq 0,
\]

means that transition \( t \) should be disabled whenever place \( p \) contains a token, or that all system states that would allow transition \( t \) to be enabled are forbidden whenever place \( p \) contains a token. The interpretation lies in the particulars of a given system and its operation. The first interpretation is called direct interpretation and the second is called indirect interpretation, both means of enforcing the constraint can be useful for different problems [Moody and Ansalklis 1998].

Furthermore, supervisory control for a DES enforces linear equality constraints or logical constraints on the system’s behavior of the form:

\[
\text{If } \mu(p) \neq 0, \text{ then } t = 0
\]

This constraint form can be applied in Petri Nets with places that have two states: either they contain a token or they do not, e.g., safe Petri Nets, and similarly all transitions can be viewed as having two states: either they will fire in the current iteration of the system’s evolution or they will not.

### 4.4.7 Logical Control Method for a DES

This document proposes a logical control method to synthesize controller nets for a DES using inhibitor arcs to design and synthesize controller nets to avoid uncontrolled execution of VL subsystems. The proposed method is a modified version of the control method presented by [Wu et al. 2001]. The proposed modifications simplify the method and synthesize non-maximally permissible controller nets that make
it compatible with the event-based control approach for Internet-based telerobotic applications.

Inhibitors arcs are Petri Net extensions introduced to model systems, such as producer-consumer systems [Murata 1989]. An inhibitor arc connects a place $p$ to a transition $t$ and is represented by a line terminating with a small circle instead of an arrowhead at the transition, as shown in Figure 4.13. The inhibitor arc disables $t$ when $p$ has a token, and enables $t$ when $p$ has no token and other (normal) input places have at least one token per arc. No tokens are moved through an inhibitor arc when $t$ fires. A class of Petri Nets with inhibitor arcs is referred to as extended Petri Nets. Inhibitor arcs add the ability to test zero, e.g., absence of tokens in a place.

![Figure 4.13: Inhibitor arc behavior](image)

Figure 4.13: Inhibitor arc behavior: (a) transition $t_1$ is able to fire because it has a token in the input place $p_1$ and none in place $p_2$. (b) Transition $t_1$ is disabled because of the token in place $p_2$.

The proposed control method will enforce logical constraints according to the following form:

$$\text{If } \mu(p_{n+1}) \neq 0, \text{ then } \sum_{i}^{n} \mu(p_i) = 0 \quad (4.17)$$

where $i < n$, $i \in Z^+$, and $n \in Z^+$. This constraint form describes the following logical constraint: if place $p_{n+1}$ has one token, then the sum of tokens in places $p_i, p_{i+1}, p_{i+2}, \ldots p_n$ must be zero. Places $p_i, p_{i+1}, p_{i+2}, \ldots p_n$ are called constrained places.

Notice that the control method is designed to have the following restrictions:

- It is only applicable to safe, ordinary Petri Nets, since the constraint form is represented as a logical constraint, as shown in Eq. 4.16.

- Constrained places are sequentially ordered in the Petri Net. The constrained places must satisfy the following conditions: $p_i^0 = o p_{i+1}$ and $p_{i+1}^0 = o p_{i+2}$ ... and $p_{n-1}^0 = o p_n$.

- The output transition of the last constrained place $p_n$ must have $p_{n+1}$ as its output place, this means that $p_n^0 = o p_{n+1}$.
4.4. Composition, Analysis and Control of a Dynamic Model

- It is non-maximally permissible, which means that the method will modify the original system behavior; this condition may occur when constraints have the form of equalities rather than inequalities [Moody and Ansalklis 1998].

The following definitions and notations are used by the method:

**Definition 4:** The place set $p_i, p_{i+1}, p_{i+2}, \ldots p_n$ described by the summation in Eq. 4.17 is named as the logical constrained place set, denoted by $L_c$, that is,

$$L_c = \{p_i|\sum_{i}^{n} \mu(p_i) = 0\}. \quad (4.18)$$

**Definition 5:** The set of input transitions for the entire logical constrained place set $L_c$ is said to be the logical input constrained transition set, denoted by $\overset{o}L_c$, that is,

$$\overset{o}L_c = \{t|t \in \overset{o}p \ \text{for} \ p \in L_c\}, \quad (4.19)$$

where $\overset{o}p$ denotes the input transition for place $p$.

**Definition 6:** The set of output transitions for the entire logical constrained place set $L_c$ is said to be the logical output constrained transition set denoted by $L_{c^o}$, that is,

$$L_{c^o} = \{t|t \in \overset{o}p \ \text{for} \ p \in L_c\}, \quad (4.20)$$

where $\overset{o}p$ denotes the output transition for place $p$.

**Definition 7:** The set of transitions denoted by $CL_{ct}$ is said to be the logical common constrained transition set if its entry $t$ satisfies:

$$t \in \overset{o}L_c \cap L_{c^o} \quad (4.21)$$

**Definition 8:** Given the logical input constrained set $\overset{o}L_c$, the set $\overset{o}L_{pure-ct} = \overset{o}L_c - CL_{ct}$ is said to be the pure logical input constrained transition set.

**Definition 9:** Given the logical output constrained set $L_{c^o}$, the set $L_{pure-ct} = L_{c^o} - CL_{ct}$ is said to be the pure logical output constrained transition set.

This method for synthesizing controller nets enforces logical constraints doing the following four steps:

1. Evaluate the related sets $L_c, \overset{o}L_c, L_{c^o}, CL_{ct}, \overset{o}L_{pure-ct}$, and $L_{pure-ct}$.
2. Construct a controller (monitor or slack) place $p_c$. The Petri Net will not satisfy the desired constraint without an external controller.
3. For each $t \in L_{pure-ct}$, draw an arc between the controller $p_c$ and the transition $t$. Let $p_c$ be the output of $t$.
4. For each $t \in \overset{o}L_{pure-ct}$ draw an inhibitor arc between the controller $p_c$ and the transition $t$. Let $t$ be the input of the inhibitor arc.
4.4.8 Behavioral Analysis of Controlled Petri Net

The composed Petri Net with controller net structures is analyzed to verify that experiment functionalities and event-based control properties are met. From the experimental point of view, VL subsystems to mount, define, and execute the experiment must be sequentially executed. This condition is translated into three requirements: first, the mount subsystem, starting from the initial state of the VL system, must be able to reach its goal state. Second, the define subsystem, starting from the mount goal state, must be able to reach its goal state. Third, the execute subsystem, starting from the define goal state, must be able to reach the VL goal state. Therefore, the VL system, starting from its initial state and passing through the mount, define, and execute experiment stages, must be able to reach its final goal state. These design requirements are formally stated with the following theorem.

**Theorem 5:** A Petri Net representing a VL system data flow executes sequentially its experiment functionalities to mount, define, and execute an experiment, if and only if, in the coverability tree of the Petri Net, there is a firable transition sequence \( \tau \) that accomplishes the following equation:

\[
M_0[\tau > M_e \Rightarrow M_0[\tau_m > M_m \text{ and } M_m[\tau_d > M_d \text{ and } M_d[\tau_e > M_e \text{ are defined}], (4.22)
\]

where the firable transition sequence concatenation is \( \tau = \tau_m + \tau_d + \tau_e \) and \( M_m \) represents the system marking that includes the goal state of the mount subsystem, \( M_d \) represents the marking that includes the define goal state, and \( M_e \) represents the marking that includes the execute goal state (VL goal state).

**Proof:**

*Necessary.* Assume a Petri Net representing a VL system data flow with a firable transition sequence \( \tau \), in a way that \( M_0[\tau > M_e \) and \( \tau = \tau_m + \tau_e \). This means that the sequence \( \tau \) overrides the define subsystem and reaches the VL goal state without defining the levels of specific independent variables by the user. Therefore, the VL system represented by the Petri Net does not accomplish the necessary stages of a true experiment.

*Sufficient.* Assume that the VL Petri Net has a sequence \( \tau = \tau_m + \tau_d + \tau_e \) and Eq. 4.22 is defined, and according to the concatenation of \( P \)-type languages \( (\tau \in T_{PN}^*), \) this implies that the VL accomplishes the necessary stages of a true experiment, which means that the mount subsystem is executed firstly from the initial state and reaches its goal state, the define subsystem is executed secondly from the mount goal state and reaches its goal state, and the execute subsystem is executed thirdly from the define goal state and reaches the VL goal state. Therefore, these three subsystems are executed sequentially. \( \Box \)

Notice that the coverability tree must be built, taking into account the composed Petri Net and its controller net structures. However, to prove the sequential execution
of VL subsystems and the event-synchronous property of the original composed Petri Net, the coverability tree has to be analyzed without considering the controller net structures. This means that the markings of the composed Petri Net in the coverability tree must behave as a safe net, as a state machine, and the \textit{mount, define, and execute} subsystems must have a transition \( t \) with at least a liveness level \( L1\text{-live} \) to reach their goal states in a sequential way.

### 4.5 Structural and Dynamic Designs Implementation

This chapter has described the development methodology for VLs. The static structure of a VL design is described using UML diagrams to give a complete and detailed description of VL components to perform specific telerobotic experiments and provide a guideline for its object-oriented implementation. Final phases of the development methodology generate a complex Petri Net, which models appropriate dynamic behavior of the VL design. These static and dynamic formal definitions model a complete VL application, as it appears in Fig. 4.14. Both UML and Petri Net formalisms offer better understanding of designed VL systems to be implemented and help to visualize and appreciate their complexity.

UML diagrams can be taken as a reference to implement, with high level programming languages using the object-oriented paradigm, such as, C++, Java, Visual Basic, and others. The Petri Net model assures that the overall control of intrinsic procedures in VL systems was successfully tested before its implementation, and can be effectively implemented in a programming language.

This development methodology can be expanded to consider other VL design aspects, such as: user management, resource sharing, collaborative experimentation, tutor guidance, and error recovery, among others. This expansion can be done because development methodology combines event-based control design and object-oriented foundations, and it can structurally and dynamically define VL applications using formal modeling notations (UML and Petri Nets).

### 4.6 Advantages of the Proposed Methodology

One of the difficulties of designing and implementing VLs for telerobotics is the lack of a generic model and a general development methodology. Both the model and methodology will avoid costly development of new VL applications from scratch [Amoretti \textit{et al.} 2003]. The proposed generic and modular model allows the developer to be aware of teleoperation strategies that are well known to those with more experience in the telerobotics research field. It is composed of generic entities that
are independent of vendor and technology. The model defines the appropriate configuration of desired processes and devices in a VL. This model is translated into the modeling language UML that allows the developer to visualize, specify, construct, and document artifacts of software-intensive systems, using an object-oriented approach [Booch et al. 1999]. This object-oriented approach is part of the software development mainstream because it has proven to be valuable in building systems in all sorts of problem domains and encompassing every degrees of size and complexity [Fowler 2002]. Therefore, the object-oriented model is integrated as a reference framework into the development methodology to produce high quality and low cost software. The reuse of software artifacts (repeated process instantiations in VL subsystems) is another way to increase software quality and reduce efforts and time in the development methodology. High quality, low cost, and reuse of software are supported by several characteristics of the object-oriented paradigm: reuse through inheritance, data hiding through encapsulation, and appropriate behavior using class polymorphism [Liberty 2000].

This proposed methodology provides a novel procedure that formally transforms the dynamic part of the reference framework into the Petri Net notation. This notation provides necessary analytical techniques to validate quantitatively and qualitatively the behavioral design of VLS based on the event-based approach for Internet-based telerobotics [Xi 1993, Elhajj et al. 2003]. Petri Nets have been used to model and
analyze all kinds of processes with applications ranging from protocols, hardware, and embedded systems to manufacturing systems, user interaction, and business processes [VanDerAalst 1998]. There are several reasons for using Petri Nets for VL modeling: formal semantics that clearly and precisely specify the behavioral of VLs; graphical nature, which brings intuitive and easy learning; expressiveness in modeling the signal flow in a VL system; analysis techniques to prove and validate event-based design requirements, which help to detect design errors before their implementation, and vendor-independence to provide an independent tool for modeling and analyzing VL processes.

4.7 Conclusions

The development methodology provides a novel approach for modeling, designing, analyzing, and validating the structural and dynamic design of VL applications for telerobotics. It takes, as its starting point, a well-defined reference framework for VL software and hardware based on a generic and modular model that joins tested approaches to deal with problems arising in Internet-based telerobotics area. This reference framework describes the generic and modular model, using multiple diagrams, that emphasize interrelationships among model elements and entities viewed from different perspectives.

The framework provides a skeleton, which is customized by the development methodology, to model necessary VL subsystems for telerobotic experimentation. It allows representation of the structure of complex VL systems in an object-oriented modeling language. Also, the methodology has formal procedures to translate the customized framework into another modeling graphical notation, which allows its behavioral definition to support its analysis and validation in a quantitative and qualitative way. Furthermore, the methodology takes into account experiment functionalities to perform a true experiment, and provides a way to validate their sequential execution to achieve experimentation goals.

This proposed methodology introduces an original approach to model the structure and dynamics of VL applications for telerobotics. Existing development approaches, summarized in Table 2.1, had not provided a complete design method that included structural and dynamic modeling. The methodology guides the developers when designing and building VLs for telerobotics over the Internet. UML and Petri Net diagrams, generated by the methodology, provide a complete VL design that is fully compatible with object-oriented programming languages.

Chapters 5 and 6 present the structural and dynamic development of two VLs for telerobotics, using the proposed methodology. Chapter 5 describes the development of a VL for mobile robotics. Chapter 6 presents the development of a VL for bilateral
teleoperation. Implementation specifications and experimental results are described in Chapter 7.
Chapter 5

Design of a Virtual Laboratory for Mobile Robotics

Two strategies allow illustrating and validating the generality of the proposed model and its development methodology: designing and developing a new experiment as a VL, or transforming an already designed computer-based experiment as a VL. In this chapter, the second choice was selected to decrease efforts and development time. The selected experiment is a collection of well known mobile robotics and computer vision techniques integrated into a stand-alone application. This application has a diversity of procedures and devices that illustrate and validate the complete model and its development theory.

The proposed model and its methodology are validated by performing the structural and dynamic development of a VL for mobile robotics experimentation. The model and its methodology are applied on an existing robotic experiment to generate a VL for mobile robotics over the Internet, as shown in Figure 5.1. The resulting VL for mobile robotics was described in [Borstel et al. 2003a].

The first methodology phase analyzes the existing experiment, its description, and its process flow chart, to generate specification tables that summarize necessary functionalities to perform a true experiment. From the experiment description, hierarchical charts are generated to group experiment functionalities, procedures, and functions.

Schemes, specifications, and charts are then translated into customized instantiations of the VL framework where class, collaboration, deployment, sequence, and statechart diagrams are generated. Experiment functionalities are now treated as separated VL subsystems. The hierarchical charts are identified as instantiations and operation methods of VL class components. Object associations are extracted from the process flow chart to generate UML collaboration diagrams. UML sequence diagrams are generated from experiment and functionality descriptions. Detailed operation of the VL design is defined through UML statecharts to represent intrinsic states of VL
Chapter 5. Design of a Virtual Laboratory for Mobile Robotics

Figure 5.1: Structural and dynamic modeling diagrams generated by the development methodology. The first methodology phase generates a process flow chart, specification tables, and hierarchical charts. The second phase produces a customized UML framework. The third phase converts framework dynamics into Petri Net formalism to analyze VL subsystems. The last phase generates a composed Petri Net to provide a validated dynamic design.

subsystems. This phase of the development methodology produces customized and detailed models of VL subsystems.

Once the static structure and internal behavior of these subsystems are defined, UML statechart diagrams are converted into Petri Nets. The methodology takes advantage of structural and behavioral properties of Petri Nets to perform a quantitative and qualitative analysis on each VL subsystem. The coverability tree analysis is used to find transition properties that are related to behavioral problems of VL subsystem designs. Generated Petri Nets are verified, and if necessary, redesigned to accomplish their goals.

The following methodology phase merges Petri Nets to generate a composed Petri Net. This composed Petri Net is analyzed to verify that experiment objectives are met. Modifications to meet experiment objectives are performed based on supervisory
control methods for a Discrete Event System (DES).

The customized UML framework and its composed Petri Net are taken as a guideline to implement a VL application using object-oriented programming languages.

5.1 Experiment Design and Functionality Analysis

The first phase of the development methodology starts performing a functionality analysis. In this case, the experiment design is based on a stand-alone application [Ponce 2002] that performs robotics path planning and applies computer vision techniques to control a Khepera minirobot [K-Team 2002] in a small and delimited workspace.

The robotic navigation experiment works in the following way: the experimental environment is set when the user indicates physical obstacle vertices on a workspace top view image displayed by a user interface. These vertices are used by the system to establish a Configuration Space ($C$-Space) in accordance with robot characteristics and its current environment [Latombe 1991]. Once the $C$-Space is established, the user indicates a point to be reached by the robot. Then, the system draws an obstacle-avoiding trajectory on the image. If the user accepts the planned trajectory, then the system makes the robot follow that path using the computer vision system.

5.1.1 Experiment Description

The experiment path planner uses a potential field method, while the path follower uses the planned path as its reference to visually track and control robot movements. Figure 5.2 describes the experiment process flow chart.

The workspace image is acquired by a camera that outputs compound video. The video signal is converted into discrete data using a video grabber board. This process generates a matrix $I$, with $320 \times 240$ integer elements on a 256-level gray scale. This $I$ matrix is displayed on the user interface as an image Bitmap $P$. When the workspace image and the robot’s shape are displayed, the user interacts with the user interface to define a $C$-Space based on the displayed image. This definition is done by using a PC mouse; the user marks on the image obstacle’s vertices $(x_v, y_v)$ and the system computes a vertex convex hull [Graham 1972] drawing a convex polygon $H_i$, which expands the robot’s radius, and displayed again on the image to represent a $C$-Obstacle $E_i$. This procedure is carried out $m$ times, once for each physical obstacle.

If the user is satisfied with this $C$-Space definition, then the system converts the displayed Bitmap $P$ into a matrix $C$ with the same size as $I$, where pixels inside every convex polygon’s area are represented by elements with high deci-
Figure 5.2: Process flow chart of the robotic navigation experiment. The C-Space is established when the user introduces the obstacle's vertices, which are processed to produce a convex hull that is expanded to generate C-Obstacles. After that, the C-Space is converted into a numeric matrix used by the Navigation Function (NF1) algorithm to generate a potential field. Then the image is processed to calculate the current robot position, and a best-first search is performed to calculate an obstacle-avoiding path. Then, the planned path is used as the reference by the control process to visually track and control robot movements to follow that path.

This matrix $C$ models a three-dimensional C-Space grid [Barranquand et al. 1992], which is used by the path planning process. The path planning process starts once the user indicates the destination point $(x_f, y_f)$ on the image. This process is based on the wavefront expansion NF1 method [Latombe 1991] and a best-first path search using minimal Manhattan distance heuristics. These procedures generate a continuous planned path $\{(x, y), \ldots\}$. A cubic spline interpolation procedure [Burden and Faires 1993, Hernandez 1995] is used to smooth the continuous planned path, which is analyzed to get slope change points to keep them in an array of $n$ sequential points $\{(x_t, y_t), \ldots\}$. Then, the path follower procedure uses this array as reference.

On the other hand, matrix $I$ is binarized and segmented to find every object $O_i$ in the image. After that, objects found are characterized, using the invariant moment method to identify a robot artificial mark [Hu 1962]. Two circles compose the robot's landmark: the bigger circle represents the robot's front while the smaller one indicates the rear. The robot position $(x_k, y_k)$ is the big circle centroid, and the robot orientation $\theta_k$ is the angle defined by a line passing through both circle centroids and
the $x$-axis of the reference frame.

Path-following procedures compare the current robot position $(x_k, y_k, \theta_k)$ with the next path point to reach $(x_t, y_t)$, e.g., an advance speed command for the left and right wheel motors $d(+v_l, +v_r)$ is sent to the robot if a specified threshold $\delta$ is smaller than the angular difference $\theta_h$ between $\theta_k$ and the angle defined by a line passing through $(x_k, y_k)$ and $(x_t, y_t)$ and the $x$-axis. An orientation speed command $d(\pm v_l, \pm v_r)$ is sent if the threshold $\delta$ is greater than $\theta_h$. A detailed description of robotics and vision techniques is presented in Appendix B.

### 5.1.2 Experiment Analysis

The first phase continues performing an analysis, based on the experiment description, to produce a complete definition of experiment functionalities. This definition details experiment procedures, their interrelationships, specifications, and internal activities. In this case, the experimental application has three main functionalities to create the C-Space (Mount), plan the obstacle-avoiding path (Define), and follow the planned path (Execute) in the robotic navigation experiment. The following functionalities have to be performed in a synchronized and sequential way.

**C-Space Creation:** This functionality encapsulates necessary procedures to transform the obstacle's vertices, marked by the user, into convex polygons to represent physical obstacles. Then, these polygons are expanded according to the robot's shape to create C-Obstacles and simulate a C-Space. This C-Space is displayed on a workspace image obtained from the image processing procedures and the camera. Note that if an already created C-Space is saved into the C-Space database, then the user is able to retrieve it. Thus, some procedures of the C-Space creation can be skipped when a saved C-Space is retrieved from the database.

**Path Planning:** Once the C-Space has been defined and displayed, the generated bitmap is converted into an integer matrix to be used as a numerical grid by the Navigation Function (NF1) algorithm. Note that the current robot position is required to perform the NF1 algorithm. The robot position is calculated using the image processing procedures to identify robot landmarks on the workspace. Then a gradient descent is performed on the filled matrix to find a path in the obstacle-free C-Space (C-Space$_{free}$). This path is smoothed by a Spline algorithm. The generated curve is displayed on the workspace image.

**Path Following:** The planned path is sent to control procedures that compare every path point with the current robot position to generate an appropriate command for the robot to reach that point. This procedure is performed until the complete trajectory is followed and the destination point is reached.
5.2 \textit{C-Space} Creation Functionality Modeling

The \textit{C-Space} Creation functionality is identified as necessary procedures to mount the robotic navigation experiment. This functionality defines several independent variables to perform the robotic experiment. It creates a specific \textit{C-Space} of the workspace according to the robot's shape.

This section illustrates the first three methodology phases, which are used to translate the \textit{C-Space} Creation functionality into UML and Petri Net modeling notations.

5.2.1 \textit{C-Space} Creation Functionality Analysis

To continue the first phase of the development methodology, a specification table and a hierarchical chart are build to describe procedures of the \textit{C-Space} Creation functionality, their specifications and functions. Table 5.1 extracts and summarizes the most significant \textit{C-Space} Creation processes, specifications, and functions. Notice that physical devices are also described as processes with specifications and functions.

Table 5.1: The most significant process specifications of the \textit{C-Space} Creation functionality.

<table>
<thead>
<tr>
<th>Process</th>
<th>Function Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>Displays GUI, image Bitmap (320 × 240 pixels) and \textit{C-Space}</td>
</tr>
<tr>
<td>Mouse</td>
<td>Generates discrete signals for vertices ((x_v,y_v)) and acceptation</td>
</tr>
<tr>
<td>Visual Display</td>
<td>Processes the image (I) and \textit{C-Space} bitmap ((P = 320 \times 240) pixels)</td>
</tr>
<tr>
<td>\textit{C-Space} Creation</td>
<td>Calculates convex hull and expands polygons (H_i) according to robot's shape to create \textit{C-Obstacles} (E_i)</td>
</tr>
<tr>
<td>User Interface</td>
<td>Gets vertex coordinates ((x_v,y_v)) and acceptation</td>
</tr>
<tr>
<td>\textit{C-Space} DB</td>
<td>Reads/writes obstacle definitions (O_i = {(x_v,y_v),...}) on file</td>
</tr>
<tr>
<td>Image Processing</td>
<td>Acquires image NTSC, generates matrix (I = {i_{320,240}}), 256 grays</td>
</tr>
<tr>
<td>Camera</td>
<td>Captures the workspace image</td>
</tr>
<tr>
<td>Robot</td>
<td>Khepera minirobot with artificial landmark</td>
</tr>
<tr>
<td>Workspace</td>
<td>Workspace surface ((73.5 \times 98) cm(^2)) with physical obstacles ((2 \times 3 \times 15) cm(^3))</td>
</tr>
</tbody>
</table>

The next step in the experiment analysis generates a hierarchical chart from the specification tables, as shown in Figure 5.3. This chart groups \textit{C-Space} Creation functionality procedures in a hierarchical manner. It relates processes with their specific
5.2. C-Space Creation Functionality Modeling

functions. This chart clearly describes relationships and dependencies among functionalities, procedures, and specific functions.

Figure 5.3: Hierarchical chart of the C-Space Creation functionality.

5.2.2 C-Space Creation Subsystem Identification

The second phase of the development methodology is the VL subsystem identification in the UML framework. Experiment functionalities are treated as subsystems and related to the VL framework in the first stage of the subsystem identification phase. This first stage models the structure of VL subsystems to create UML class and deployment diagrams. This stage relates specification tables and hierarchical charts with UML class and deployment diagrams by taking as reference the VL definition provided in the UML framework.

Hierarchical charts group VL subsystem procedures found in the experiment analysis. These procedures are identified as instantiations of VL classes and their functions are identified as class operation methods, based on the VL framework definition. This identification generates a model-based distribution of procedures between Guest and Host systems. This distribution is defined in the UML deployment diagram. Figure 5.4 shows instantiations and associations of identified VL classes to describe the C-Space Creation subsystem.

In Figure 5.4, the Guest system class is composed of the Guest computer, Guest software, and Output Device subclasses. The Guest computer is instantiated by a computer, which will be accessed by the user and will contain Guest software. In this case, the Input Device is an important component that is instantiated by the PC Mouse. The Output Device subclass is instantiated by the Monitor with just one operation method: display(). Notice that the Output Device class is instantiated in the three VL subsystems to provide visual feedback regarding the experiment.

In the same way, the Guest software class is composed of the Display Process, User Interface, C-Space Creation, C-Space Database, and Communication class in-
stantiations. Each procedure is extracted from the specification table and hierarchical chart and is identified as an instantiation of the Guest Software subclasses, based on the VL reference framework described in Chapter 3 and Appendix A. The Display Process is identified as the instantiation of the VL Sensory Conversion subclass, since this process converts discrete sensory information into visual sensory information. The Display Process has two operation methods: displayImage() and displayCspace(). The Input Processing subclass is instantiated by the User Interface, which has two operation methods: getVertices() and getAcceptation(). The User Interface processes discrete information from the PC Mouse to define obstacle vertex coordinates and user commands that are introduced into the C-Space Creation procedures. The Simulation subclass is related to C-Space Creation procedures. The C-Space Creation instantiation has functions that allow to identify it as a Simulation class: convexHull() and expandObs(). The C-Space Creation instantiation produces a simulated model of the robot workspace and defines a C-Space according to the robot's shape. The Settings/Tasks Database is instantiated by the C-Space Database, which has two essential operation methods: read() and write(). Identified VL classes and their operations give the static structure of the experiment to transform it into a VL application. In this case, the communication class is not found in the experiment analysis since the experiment was originally designed as a stand-alone computer system to perform the robotic experiment in a laboratory facility. Therefore, a communication class is introduced and instantiated by a TCP/IP communication process. The VL application will be designed to communicate in a peer-to-peer mode through the Internet. Thus, the communication instantiation considers two basic operation methods: transmit() and receive().
On the other side, the *Host* system is composed by the *Host* Computer, Robot, Workspace, *Host* Software, Camera, and Communication instantiations. The *Host* computer is the computer that will execute the *Host* Software procedures to remotely perform the experiment. The robot is identified as a Mechanism and Control Device class. It executes commands to navigate in the workspace and reach a position specified by the user. The physical workspace is identified as an instantiation of the Remote Environment class. The camera instantiation is identified as an instantiation of the Sensor class. This class has one operation method: `capture()` the image of the workspace and robot.

The Image Processing procedure is identified as a component of the *Host* Software class. Image Processing instantiates the Sensed Data Processing class; it has one operation method: `Acquisition()` of the image from the camera. The Communication instantiation is introduced to create a communication link between *Guest* and *Host*.

The subsystem identification phase produces a natural distribution of procedures and devices into two systems: *Guest* and *Host*. This distribution is also defined by the UML deployment diagram shown in Figure 3.9.

The second stage of the subsystem identification phase is the VL subsystem dynamic modeling. This dynamic modeling procedure extracts relationships among identified VL classes, messages sent between them, and their execution sequence from the process flow chart and experiment description. The UML collaboration diagram of the *C-Space* Creation subsystem, with identified instantiations and interrelationships, is shown in Figure 5.5.

The UML collaboration details instantiation interrelationships. The Workspace provides the camera a workspace image. The workspace image is captured by the camera and discretized as an Image Signal. This Image Signal is received by the Image Processing and is transformed into discrete Image Data. The Image Data is transmitted through the Internet by the Communication instantiation. The Display Process receives the Image Data and generates an appropriate Image signal for the Monitor. The Monitor displays the Image to the User. The User introduces analog Inputs, which are discretized by the Mouse, to indicate obstacle Vertices in the User Interface. These vertices are processed by the *C-Space* Creation instantiation to create *C-Obstacle* definitions. These definitions are introduced into the Display Process to produce a bitmap of the *C-Space*. This *C-Space* is translated as Signals for the Monitor to be Displayed. Image requests are automatically generated by the *C-Space* Creation process. These requests are transmitted by the Communication instantiation through the Internet, and received by the Image Processing procedure that acquires a new Image Signal from the Camera. This new image is transmitted to the Display Process and displayed by the Monitor, and the iteration cycle is repeated until the User accepts the *C-Space*. Once the *C-Space* is created, the User intro-
duces analog Inputs via the Mouse to activate User Interface Buttons, which are used to accept the C-Space and save or retrieve obstacle definitions of the C-Space Database.

The second stage of the subsystem identification step also describes VL subsystem execution sequences. A detailed UML sequence diagram is described in Figure 5.6. It uses extensions for the UML sequence diagrams [Jeng and Lu 2002]. In this case, just the extension for choice was used and is represented as a dashed line with two hollowed endpoints. These hollowed endpoints are drawn on the message arrows on the right side of the lifeline (vertical dotted line) of the C-Space Creation and User instantiations. The UML sequence diagram describes subsystem dynamics from a time perspective. The UML sequence and collaboration diagrams have a similar semantics, but a different syntax.

The third stage of the subsystem identification phase generates UML statechart diagrams to give a detailed description of the C-Space Creation subsystem operation, based on the previously-generated definitions. This statechart diagram describes operations and internal states produced by the interaction of VL class instantiations.

Statecharts describe operation methods (transitions) as arrows connecting rounded boxes. These rounded boxes represent internal states of VL subsystems as a result of the last performed operation. These diagrams have their foundations on finite state machines having a similar behavior. In statechart diagrams, a filled circle represents the initial state of the system and a circle with another filled circle inside represents the final state. Figure 5.7 describe internal states for the C-Space Creation subsystem.

The state diagram shown in Figure 5.7 is divided by dashed boxes to relate
transitions and internal states to class instantiations. The represented instantiations are listed with numbers between parentheses inside dashed boxes.

In the UML statechart diagram that represents the C-Space Creation subsystem, the communication class is instantiated by two operations: transmission and reception. A state in the middle of both operations is a standard in the telecommunication protocol design. This state is known as the buffer full state [Murata 1989]. Buffer full states represent the TCP transmission memory buffer where data is stored until a minimal memory segment size (mss) is completed or a time-out is reached [Comer 2000]. The communication instantiation is created to transform the experiment into a VL. This communication class instantiation can be taken as a division between Guest and Host.

However, several simplifications were made in the C-Space Creation statechart, for instance, the Get Vertices transition, and the Obstacle's Vertices Defined state represent operation methods and internal states occurring when the user interacts with the PC Mouse and User Interface to introduce obstacles' vertices. The Process and Display transition represents operation methods performed by the Image Process and Monitor instantiations.
5.2.3 UML-Petri Net Conversion and Analysis

The UML-Petri Net conversion phase transforms UML statechart diagrams into the Petri Net formalism to model and analyze VL subsystem dynamics. The conversion function \( G(ST) \) formally transforms a UML statechart diagram \( ST \) into an ordinary Petri Net \( PN, G(ST) = PN \). Figure 5.8 is the Petri Net representation of the \( C\)-Space Creation subsystem.

Once the \( C\)-Space Creation subsystem is transformed into the Petri Net shown in Figure 5.8, the Petri Net is analyzed to validate its design as a stable event-based subsystem. This analysis takes into account several behavioral properties of Petri Nets related to the event-synchronization property of event-based VL systems. From the Petri Net diagram in Figure 5.8 one can deduce that a self loop in \( p_9 \) and all its arc weights are 1s. Therefore, it is a non-pure and ordinary Petri Net graph. Furthermore, some places have more than one output arc in its structure, such as \( p_5 \) and \( p_9 \). This implies that the Petri Net has conflicts in its structure. Therefore, it is a structure exhibiting nondeterminism.

From the coverability tree analysis of the \( C\)-Space Creation subsystem shown in Figure 5.9, one can deduce that the system is 1-bounded (is safe), all transitions are
5.2. C-Space Creation Functionality Modeling

firables (they are coverable in the coverability tree), and there is a firable transition sequence $\tau$ that allows the potential firability of transition $t_{11}$ from the initial marking $M_0$, which implies that $t_{11}$ is L1-live. When transition $t_{11}$ is fired in the coverability tree, the C-Space Creation subsystem reaches its goal state ($p_{11}$, C-Space defined).

Furthermore, all transitions in the C-Space Creation Petri Net have one input and one output place. It implies that a procedure cannot generate more than one signal. Therefore, there exists only one signal instantiation at any instant in the subsystem. Thus, the C-Space Creation subsystem behaves as a state machine automaton; 1-bounded, L1-live, and state machine properties imply that the system is event-synchronized.

Figure 5.8: Petri Net diagram of the C-Space Creation subsystem.
Figure 5.9: Coverability tree representing the C-Space Creation subsystem internal states.
5.3 Path Planning and Path Following Functionalities Modeling

Appendix D, section D.1 summarizes the modeling phases of Path Planning and Path Following functionalities. The functionality modeling describes the first three phases of the VL development methodology: experiment functionality analysis, experiment subsystem identification, and experiment dynamics conversion for analysis.

Mobile robotics experiment modeling is presented in Appendix D, which includes specification tables and hierarchical charts. These descriptions are translated into the UML graphical notation to generate class, collaboration, sequence, and statechart diagrams. The framework dynamics is extracted, based on the statechart diagram, and converted into Petri Nets. These Petri Nets are analyzed to relate their behavioral properties with event-based control properties, using the coverability tree method. The event-synchronization property is validated via liveness, boundedness, and automaton characteristics of Path Planning and Path Following Petri Nets.

Analytical results in the last functionality modeling phases will conclude that the three VL subsystems can accomplish their goals. They are well-designed and can perform their specific tasks. These subsystems are represented by Petri Nets with the necessary behavioral properties to assure their own stability and synchronization as separate event-based applications. However, a VL system must integrate necessary functionalities to perform independent and dependent variable manipulation and experiment execution. Therefore, these subsystems must be merged into one system to be considered a VL.

5.4 Petri Net Synchronous Composition

Based on synchronous composition of transitions and parallel place fusion reduction of Petri Nets, these subsystems are merged into a synchronous composed Petri Net. The first composition $PN_1 \parallel PN_2 = PN_3$ merges the $C$-Space Creation $PN_1$ and the Path Planning subsystems $PN_2$. Then $PN_3 \parallel PN_4 = PN_5$, where the resulting composed Petri Net $PN_3$ is merged with the Path Following subsystem $PN_4$; the final composed Petri Net $PN_5$ is shown in Figure 5.10.

Note that the numeric labeling starts using as reference the first subsystem to create a new joined numeric labeling for analytical purposes. Since the merged subsystems must be executed in a sequential way, the initial marking of the composed Petri Net is always the initial marking of the first merged subsystem ($C$-Space Creation). In this case, a token is placed in place $p_1$ to represent the initial system state: camera ready. Textual labels are not modified because they are the reference for the synchronous transition composition and the parallel place fusion.
Table 5.2: Joined place numeric labeling and its description.

<table>
<thead>
<tr>
<th>Place</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>camera ready</td>
</tr>
<tr>
<td>$p_2$</td>
<td>image acquired</td>
</tr>
<tr>
<td>$p_3$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_4$</td>
<td>image received</td>
</tr>
<tr>
<td>$p_5$</td>
<td>image displayed</td>
</tr>
<tr>
<td>$p_6$</td>
<td>vertices defined</td>
</tr>
<tr>
<td>$p_7$</td>
<td>obstacle defined</td>
</tr>
<tr>
<td>$p_8$</td>
<td>C-Obstacle built</td>
</tr>
<tr>
<td>$p_9$</td>
<td>C-Obstacle displayed</td>
</tr>
<tr>
<td>$p_{10}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{11}$</td>
<td>C-Space defined</td>
</tr>
<tr>
<td>$p_{12}$</td>
<td>matrix created</td>
</tr>
<tr>
<td>$p_{13}$</td>
<td>destination defined</td>
</tr>
<tr>
<td>$p_{14}$</td>
<td>matrix created</td>
</tr>
<tr>
<td>$p_{15}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{16}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{17}$</td>
<td>image processing enable</td>
</tr>
<tr>
<td>$p_{18}$</td>
<td>image binarized</td>
</tr>
<tr>
<td>$p_{19}$</td>
<td>object array filled</td>
</tr>
<tr>
<td>$p_{20}$</td>
<td>robot mark recognized</td>
</tr>
<tr>
<td>$p_{21}$</td>
<td>robot position calculated</td>
</tr>
<tr>
<td>$p_{22}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{23}$</td>
<td>position received</td>
</tr>
<tr>
<td>$p_{24}$</td>
<td>path created</td>
</tr>
<tr>
<td>$p_{25}$</td>
<td>smooth path created</td>
</tr>
<tr>
<td>$p_{26}$</td>
<td>path planned</td>
</tr>
<tr>
<td>$p_{27}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{28}$</td>
<td>control performed</td>
</tr>
<tr>
<td>$p_{29}$</td>
<td>destination reached</td>
</tr>
</tbody>
</table>
5.4. Petri Net Synchronous Composition

Figure 5.10: Composed Petri Net that represents the merged VL subsystems: C-Space Creation, Path Planning, and Path Following
Table 5.3: Joined transition numeric labeling and its description.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>capture and acquire image</td>
</tr>
<tr>
<td>$t_2$</td>
<td>transmit image</td>
</tr>
<tr>
<td>$t_3$</td>
<td>receive image</td>
</tr>
<tr>
<td>$t_4$</td>
<td>process and display</td>
</tr>
<tr>
<td>$t_5$</td>
<td>get vertices</td>
</tr>
<tr>
<td>$t_6$</td>
<td>calculate convex hull</td>
</tr>
<tr>
<td>$t_7$</td>
<td>expand convex hull</td>
</tr>
<tr>
<td>$t_8$</td>
<td>process and display C-Space</td>
</tr>
<tr>
<td>$t_9$</td>
<td>transmit image request</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>receive image request</td>
</tr>
<tr>
<td>$t_{11}$</td>
<td>write C-Space</td>
</tr>
<tr>
<td>$t_{12}$</td>
<td>read obstacle definition</td>
</tr>
<tr>
<td>$t_{13}$</td>
<td>get acceptance</td>
</tr>
<tr>
<td>$t_{14}$</td>
<td>convert C-Space to matrix</td>
</tr>
<tr>
<td>$t_{15}$</td>
<td>get destination</td>
</tr>
<tr>
<td>$t_{16}$</td>
<td>perform NF1</td>
</tr>
<tr>
<td>$t_{17}$</td>
<td>transmit get ready</td>
</tr>
<tr>
<td>$t_{18}$</td>
<td>receive get ready</td>
</tr>
<tr>
<td>$t_{19}$</td>
<td>transmit enable image processing</td>
</tr>
<tr>
<td>$t_{20}$</td>
<td>receive enable image processing</td>
</tr>
<tr>
<td>$t_{21}$</td>
<td>perform binarization</td>
</tr>
<tr>
<td>$t_{22}$</td>
<td>perform segmentation</td>
</tr>
<tr>
<td>$t_{23}$</td>
<td>perform characterization</td>
</tr>
<tr>
<td>$t_{24}$</td>
<td>calculate position</td>
</tr>
<tr>
<td>$t_{25}$</td>
<td>transmit position</td>
</tr>
<tr>
<td>$t_{26}$</td>
<td>receive position</td>
</tr>
<tr>
<td>$t_{27}$</td>
<td>perform gradient descent</td>
</tr>
<tr>
<td>$t_{28}$</td>
<td>perform Spline algorithm</td>
</tr>
<tr>
<td>$t_{29}$</td>
<td>process and display path</td>
</tr>
<tr>
<td>$t_{30}$</td>
<td>transmit path</td>
</tr>
<tr>
<td>$t_{31}$</td>
<td>receive path</td>
</tr>
<tr>
<td>$t_{32}$</td>
<td>perform control</td>
</tr>
<tr>
<td>$t_{33}$</td>
<td>execute command by robot</td>
</tr>
<tr>
<td>$t_{34}$</td>
<td>stop following</td>
</tr>
</tbody>
</table>
5.4.1 Composed Petri Net Analysis

The composed Petri Net shown in Figure 5.10 inherits several properties from merged VL subsystems. Non-determinist, boundeness (safe), and liveness properties are incorporated into the composed Petri Net. Unfortunately, when VL subsystems are merged, they increase non-determinism in the composed system. Therefore, the composed Petri Net must be analyzed to verify if merged VL subsystems behave appropriately.

The coverability tree analysis of the composed Petri Net shown in Figure 5.11 is helpful to determine which Petri Net segments should be disabled at certain experiment stages (mount, define, execute). Several uncontrolled events were detected from the coverability tree analysis. Necessary conditions to avoid unwanted events in the system are described by the following statements:

1. Image processing places \{p_{16}, p_{17}, p_{18}, p_{19}, p_{20}, p_{21}\} are not necessary in the C-Space Creation stage of the system. The image processing segment has to be enabled when the system is in the Path Planning stage. This condition can be described as an if-then statement: if the C-Space is still not defined, this means
that \(t_{13}\) is not fired and \(\mu(p_{11}) = 0\). Then transmission of the signal to enable image processing \(t_{19}\) must not be fired.

2. When the user has finished the C-Space Creation (\(p_{11}, \text{C-Space defined}\)), it is necessary to disable part of the system to avoid returning into the C-Space Creation stage. This condition can be described by the following statement: if the C-Space is defined, this means that \(t_{13}\) was fired and \(\mu(p_{11}) = 1\), then the C-Space Creation places \(\{p_6, p_7, p_8, p_9\}\) must be permanently disabled \(\{\mu(p_6) = 0, \mu(p_7) = 0, \mu(p_8) = 0, \mu(p_9) = 0\}\).

3. When the planned path reception \(t_{31}\) is fired, the system changes its marking to place \(p_{28}\) (control performed), and starts the Path Following stage. Then, it is necessary to disable the net segment that plans the path. This condition can be described by the following statement: if the planned path is received, this means that \(t_{31}\) was fired and \(\mu(p_{28}) = 1\), and the Path Planning places \(\{p_{22}, p_{23}, p_{24}, p_{25}, p_{26}, p_{27}\}\) must be permanently disabled \(\{\mu(p_{22}) = 0, \mu(p_{23}) = 0, \mu(p_{24}) = 0, \mu(p_{25}) = 0, \mu(p_{26}) = 0, \mu(p_{27}) = 0\}\).

4. When the C-Space Creation and Path Planning stages are performing their functions, it is necessary to disable the path following procedures. The condition statement is: if the planned path reception transition \(t_{31}\) is still not fired, then transition \(t_{32}\) to perform control must be disabled.

These statements are then transformed into logical constraints that are used to synthesize Petri Net structures to avoid unwanted (forbidden) states of the system.

### 5.4.2 Petri Net Control Design

The composed Petri Net is complemented with controller nets to avoid forbidden states. The following control constraints were identified in the Petri Net analysis:

\[
\text{If } t_{13} = 0, \quad \text{then } t_{19} = 0 \quad (5.1)
\]

\[
\text{If } \mu(p_{11}) \neq 0, \quad \text{then } \mu(p_6) + \mu(p_7) + \mu(p_8) + \mu(p_9) = 0 \quad (5.2)
\]

\[
\text{If } \mu(p_{28}) \neq 0, \quad \text{then } \mu(p_{22}) + \mu(p_{23}) + \mu(p_{24}) + \mu(p_{25}) + \mu(p_{26}) + \mu(p_{27}) = 0 \quad (5.3)
\]

\[
\text{If } t_{31} = 0, \quad \text{then } t_{32} = 0 \quad (5.4)
\]

The first and fourth logical constraints are synthesized by applying a direct implementation procedure, and the other two are synthesized by using the proposed logical control method.

Since the first constraint (Eq. 5.1) defines that, while \(t_{13}\) does not fire, then \(t_{19}\) must not fire. It is added to a controller net with a place \(p_{30}\) (enable image processing) as input place for \(t_{19}\) that dependents of \(t_{13}\) fire to obtain a token. This means that \(\alpha t_{19} = \{p_5, p_{30}\}\) and \(t_{13}^0 = \{p_{11}, p_{30}\}\). Thus, if \(t_{13}\) fires, then \(\mu(p_{30}) = 1\) and \(\mu(p_{11}) = 1\);
5.4. Petri Net Synchronous Composition

if $\mu(p_{30})$ holds until $\mu(p_6) = 1$, then both markings enable $t_{19}$. Once the C-Space is defined, it is necessary to enable the image processing net segment for Path Following procedures. This means that it is necessary to maintain able-to-fire $t_{19}$. Therefore, $t_{19}$ must generate a token for $p_{30}$ to be able to fire. Therefore, $t_9^o = \{p_{16}, p_{30}\}$, creating a self loop in place $p_{30}$, as shown in Figure 5.12.

Figure 5.12: Segment of the composed Petri Net with a controller net to enforce the first constraint in a dashed box.

Using the proposed method to synthesize the controller net to enforce the second constraint (Eq. 5.2):

1. Evaluate the sets $L_c$, $L_c^o$, $L_c^{o,ct}$, $L_{pure-ct}$, and $L_{pure-ct}^o$.

   \[ L_c = \{p_6, p_7, p_8, p_9\} \quad (5.5) \]
   \[ L_c^o = \{t_5, t_6, t_7, t_8, t_{11}, t_{12}\} \quad (5.6) \]
   \[ L_c^{o,ct} = \{t_6, t_7, t_8, t_{11}\} \quad (5.7) \]
   \[ C_{Lct} = \{t_6, t_7, t_8, t_{11}\} \quad (5.8) \]
   \[ L_{pure-ct}^o = \{t_5, t_{12}\} \quad (5.9) \]
   \[ L_{pure-ct}^o = \{t_{13}\} \quad (5.10) \]

2. Construct a controller place $p_c$. 
3. Draw an arc between place $p_c$ and transition $t_{13}$. Let $p_c$ be the output of $t_{13}$.

4. Draw an inhibitor arc between $p_c$ and for each transition in the set $t_5, t_{12}$. Let $t_5$ and $t_{12}$ be the inhibitor arc outputs. Place $p_c$ is named as $p_{31}$ in the composed Petri Net, as shown in Figure 5.13.

![Petri Net Diagram](image)

Figure 5.13: Segment of the composed Petri Net with a controller net to enforce the second constraint, which is enclosed in a dashed box.

Using the proposed method to synthesize the controller net for the third constraint (Eq. 5.3):

1. Evaluate the sets $C_L, \overset{o}{C_L}, C_L^\circ, C_{pure-Lt}$, and $C_{pure-Lt}^\circ$.

   \[
   L_c = \{p_{22}, p_{23}, p_{24}, p_{25}, p_{26}, p_{27}\} \\
   \overset{o}{L_c} = \{t_{26}, t_{27}, t_{28}, t_{29}, t_{30}\} \\
   L_c^\circ = \{t_{26}, t_{27}, t_{28}, t_{29}, t_{30}, t_{31}\} \\
   C_{L_{ct}} = \{t_{26}, t_{27}, t_{28}, t_{29}, t_{30}\} \\
   \overset{o}{L_{pure-ct}} = \{t_{23}\} \\
   L_{pure-ct}^\circ = \{t_{31}\}
   \]
2. Construct a controller place $p_c$.

3. Draw an arc between place $p_c$ and transition $t_{31}$. Let $p_c$ be the output of $t_{31}$.

4. Draw an inhibitor arc between $p_c$ and $t_{25}$. Let $t_{25}$ be the inhibitor arc output.

Place $p_c$ is named as $p_{32}$ in the controlled Petri Net shown in Figure 5.14.

![Figure 5.14: Segment of the composed Petri Net with the controller net to enforce the third constraint, which is enclosed in a dashed box.](image1)

Since the fourth constraint (Eq. 5.4) defines that, when $t_{31}$ does not fire, $t_{32}$ must not fire. It is added to a controller net with a place ($p_{33}$, enable control) as input place for $t_{32}$ that dependents of $t_{31}$s fire to obtain a token. This means that $\delta t_{32} = \{p_{21}, p_{33}\}$ and $t_{31}^e = \{p_{28}, p_{32}, p_{33}\}$. Thus, if $t_{31}$ fires, then $p_{28}$, $p_{32}$, and $p_{33}$ receive one token. If $\mu(p_{33}) = 1$ holds until $\mu(p_{21}) = 1$, then both markings fire $t_{32}$. It is necessary to maintain able-to-fire $t_{32}$ to perform the path following. Therefore, $t_{32}$ must generate a token for $p_{33}$ to be able to fire. Therefore, $t_{32}^e = \{p_{28}, p_{33}\}$, creating a self loop in place $p_{33}$, as shown in Figure 5.15. The final composed Petri Net with controller nets is shown in Figure 5.16.

![Figure 5.15: Segment of the composed Petri Net with the controller net to enforce the fourth constraint, which is enclosed in a dashed box.](image2)
Figure 5.16: Composed Petri Net with controller net structures to enforce four constraints.
5.4.3 Final Dynamic Analysis

The composed Petri Net is analyzed to verify that experiment specifications are met. The coverability tree of the composed Petri Net is shown in Figure 5.17. It was done taking into account the four added places of controller net structures that are listed in Table 5.4.

<table>
<thead>
<tr>
<th>Place</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_{30} )</td>
<td>enable image processing (first controller net)</td>
</tr>
<tr>
<td>( p_{31} )</td>
<td>disable C-Space Creation (second controller net)</td>
</tr>
<tr>
<td>( p_{32} )</td>
<td>disable planner (third controller net)</td>
</tr>
<tr>
<td>( p_{33} )</td>
<td>enable control (fourth controller net)</td>
</tr>
</tbody>
</table>

From the coverability tree analysis shown in Figure 5.17, one can deduce that each VL subsystem reaches its goal state in a sequenced and synchronized way. This means that there are three firable transition sequences \( \{ \tau_1, \tau_2, \tau_3 \} \), such that:

\[
\tau_1 = \{ t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_{13} \} \quad (5.17)
\]

\[
\tau_2 = \{ t_{14}, t_{15}, t_{16}, t_{17}, t_{18}, t_1, t_2, t_3, t_4, t_{19}, t_{20}, t_{21}, t_{22}, t_{23}, t_{24}, t_{25}, t_{26}, t_{27}, t_{28} \} \quad (5.18)
\]

\[
\tau_3 = \{ t_{30}, t_{31}, t_{33}, t_1, t_2, t_3, t_4, t_{19}, t_{20}, t_{21}, t_{22}, t_{23}, t_{24}, t_{32}, t_{34} \} \quad (5.19)
\]

and their concatenation \( \tau = \tau_1 + \tau_2 + \tau_3 \) allows:

\[
M_o[\tau > M_3 \Rightarrow M_o[\tau_1 > M_1] \text{ and } M_1[\tau_2 > M_2] \text{ and } M_2[\tau_3 > M_3] \text{ are defined}, \quad (5.20)
\]

where \( M_1 \) represents the marking vector of the C-Space Creation subsystem that includes \( \mu(p_{11}) = 1 \) (C-Space defined) and the other place markings of the composed net are equal to zero \( \{ \mu(p_1) = 0, \ldots, \mu(p_{28}) = 0 \} \), taking no care regarding controller net markings \( \{ \mu(p_{30}), \mu(p_{31}), \mu(p_{32}), \mu(p_{33}) \} \). \( M_2 \) represents the marking vector of the Path Planner subsystem which includes \( \mu(p_{26}) = 1 \) (path planned), taking into account the previous marking considerations. \( M_3 \) is the marking of the Path Following subsystem that includes \( \mu(p_{29}) = 1 \) (destination reached), taking into account the previous marking considerations. Notice that \( p_{29} \) is the VL goal state.

Without taking into account the marking of the controller net places \( \{ p_{30}, p_{31}, p_{32}, p_{33} \} \), the composed Petri Net behaves as a safe net [Murata 1989], where all its transitions are firables and have one input and output place, as a state machine automaton. Thus, it means that there exist only one signal instantiation at any instant in the VL system. Furthermore, \( t_{34} \) (stop following) is L1-live, and according to Eq. 5.20, \( p_{29} \) is reachable from \( M_o \). This implies that the VL goal state is also reachable. Therefore, the controlled and composed Petri Net is event-synchronized and performs sequentially and correctly its functionalities to mount, define, and execute the experiment.
Figure 5.17: Coverability tree of the composed Petri Net with control structures.
5.5 VL System Implementation

The composed Petri Net with controller nets has appropriate boundeness, liveness, and event-synchronous properties. Therefore, it is a stable, event-based telerobotic design for the Internet [Elhajj et al. 2003]. The final analysis step finishes the development methodology. The VL system can be implemented using UML diagrams that define its static structure and help to construct the system in an object-oriented language, such as Java or C++. The UML defines VL classes and their operation methods to be straightly implemented in an object-oriented programming language.

The Petri Net diagram tells the developer the necessary control that must be introduced into the software to mount, define, and execute the experiment. Typically, public variables and class messages in the programming code introduce control to enable or disable operation methods in different stages of the experiment.

5.6 VL Development Conclusions

This chapter presented the structural and dynamic design of a VL for mobile robotics using the development methodology. A stand-alone computer-based experiment for robotic navigation was selected to illustrate and validate the proposed methodology.

In the first methodology phase, the experiment description and its process flow chart were analyzed to generate specification tables and hierarchical charts for experiment functionalities. These tables and charts summarize processes and their function specifications. In this analysis, three main functionalities were recognized to mount, define, and execute the experiment for robotic navigation: C-Space Creation, Path Planning, and Path Following.

The second phase identified these main functionalities in the UML reference framework. In this phase, experiment functionalities are treated as VL subsystems and related to the VL framework. Grouped processes were identified as instantiations of class components to integrate a VL application, and their functions and specifications were identified as class operation methods and attributes. The reference framework was repeatedly customized according to experiment specifications to produce several UML diagrams with different perspectives of the VL subsystems.

In the third methodology phase, the VL subsystem dynamics was extracted. This phase converted statechart diagrams into Petri Nets and performed a quantitative and qualitative analysis to validate Petri Net behavioral properties that are related to event-based design requirements.

In the fourth phase, generated Petri Nets, which represent VL subsystems, were
merged into a composed Petri Net. After that, the composed Petri Net was analyzed to verify that experiment functionalities are appropriately executed. Since the composed Petri Net did not behave as desired, then several controller nets were synthesized to synchronize a sequential execution of the VL subsystems.

Finally, the customized framework and its composed Petri Net were taken as a guideline to implement the VL application using an object-oriented programming language, such as Java. UML diagrams allow visualization of the complexity of the software structure, and provide a detailed design that is fully compatible with the object-oriented paradigm. The use of this paradigm allows the developer to produce high quality software in a short time and to reuse software to reduce programming efforts.

On the other hand, Petri Nets provide a comprehensive design of the flow control among software classes. This design is fully compatible with Java event-listener class interfaces for user interface implementation, messages among classes, and public Boolean variables for control flags.

A detailed description of robotics and vision techniques, which were implemented in the VL for mobile robotics are presented in Appendix B. The VL setup and experimental results are described in Chapter 7.
Chapter 6

Design of a Virtual Laboratory for Bilateral Teleoperation

The proposed model and its methodology are used to develop a new VL for teleoperation experimentation over the Internet. This VL development takes advantage of already-designed software modules used to implement the VL for mobile robotics. It reuses object-oriented software classes as generic modules to create and build a new experiment design for Internet-based teleoperation with force reflection using an event-based control approach.

This chapter introduces a real-time control to teleoperate a nonholonomic minirobot (Khepera). It gives a detailed description of the proposed real-time control and presents its development as a VL application. The experiment analysis and VL framework identification methodology steps are described to provide a customized VL design defined by UML diagrams. Then, generated UML statechart diagrams are converted into Petri Nets to model experiment behavior and perform its analysis. These Petri Nets are merged in a composed net, which is analyzed to validate event-based design requirements. Controller nets are synthesized and integrated into the composed Petri Net to regulate its behavior. Finally, these UML diagrams and the composed Petri Net are used to guide the VL implementation. The resulting VL system for bilateral telerobotics was described in [Borstel and Gordillo 2004a].

6.1 The Teleoperation Experiment

Sensory information in real time is always desirable to increase the user's certainty in Internet-based teleoperation experiments. Visual feedback is commonly used in most of the Internet-based telerobotic systems reported in the literature [Michel et al. 2000, Hirukawa and Hara 2000, Taylor and Dalton 2000].

Recent Internet-based teleoperation applications, using the event-based control and planning theory introduced in [Xi 1993], successfully sent sensory information in real time, such as force, temperature, and video, dealing with unpredictable packet
delays, lost packets, and disconnects [Lo et al. 2003, Elhajj et al. 2003]. However, these applications used separate videoconference systems to send visual sensory information, and did not use an integrated system to provide a visual feedback in conjunction with other types of sensory information. Furthermore, they did not use visual information to obtain other types of sensory feedback.

An event-based controller for Internet-based bilateral teleoperation of a non-holonomic differential-driven mobile robot is proposed in this chapter. The controller uses a visual sensor to provide sensory information in real time. The teleoperation system makes use of visual information to generate haptic and visual sensory information that corresponds to a robot's status in the workspace. The proposed system will allow the user to control the robot for navigation and avoiding obstacles in the workspace. It combines the event-based controller with computer vision and potential field techniques to control the robot.

In this experiment, the user indicates the obstacles' vertices on a workspace image and the system computes a C-Space, which is converted into a numeric potential field grid where the cell values represent spatial information in a similar manner as occupancy grids [Elfes 1989]. This numeric grid defines three navigation spaces (forbidden, constrained, and free), as shown in Figure 6.1. Navigation spaces have cells with different probability values being occupied by an obstacle $P(\text{obs})$,

$$P(\text{obs}) = \begin{cases} 
1 & \text{forbidden space} \\
\frac{1}{f(d)} & \text{constrained space} \\
0 & \text{free space}
\end{cases} \quad (6.1)$$

where $f(d)$ is a linear function of distance $d$ between the robot and forbidden space (the obstacle). The computer vision system calculates the robot's position in real time, which is used as a reference to perform several measures on the numeric potential field grid. These measures are combined to generate a repulsive virtual force around obstacles. Furthermore, the system will provide real-time visual information regarding the robot's current position in the remote environment by drawing a graphic mark on the workspace images displayed by the user interface.

### 6.2 Real-Time Control

The numeric potential field is set when the user indicates physical obstacle vertices on a top view image of the workspace. These vertices are used to compute a C-Space. The C-Space is then converted into a numeric potential field grid, which is used to generate a repulsive virtual force when the robot approaches an obstacle. The virtual force vector is obtained, based on proximity and direction to obstacles. Proximity and direction are calculated from several measures taken from the numeric potential field grid, based on the robot's current position obtained from the computer vision system.
6.2. Real-Time Control

Obstacle's Vertices
Robot Landmarks
Workspace Image
Processed Image
C-Obstacle
C-Space
Forbidden Space
Potential Field
Constrained Space
Potential Field Measures

Figure 6.1: The numeric potential field takes the C-Space as a reference. Measures on the numeric potential field are taken using the robot's current position in the workspace as a reference.

This virtual force is converted into a velocity value and introduced into the event-based controller to generate a velocity tracking error. This tracking error is fed to the user as a force feedback to indicate proximity and direction to obstacles. Furthermore, a graphic mark is also drawn in real time on the image to indicate the robot's position.

6.2.1 Control Design

The teleoperation system design is based on two elements, as seen in Figure 6.2: the User Computer containing one or more user interface applications to allow the user to teleoperate the Khepera minirobot as slave device. In the other hand, the Laboratory Computer contains the necessary applications to remotely operate the slave device. The User Computer has connected a joystick [Microsoft 2004] and shows a user interface displaying a workspace image. The robot and a camera are connected to the Laboratory Computer. Each block will be described in detail and all variables will be defined in Table 6.1 and referenced to \( s \), where \( s \) is the event, \( s \in \{1,2,...,n\} \) and represents the number of commands issued by the user. System stability and synchronization result from the use of a non-time based variable \( s \) as reference [Xi 1993].

**Image Acquisition.** A top view of the workspace is acquired by a camera and converted into a discrete image using a video frame grabber. This process generates a matrix \( I(s) \) of integer elements, using a 256-level gray scale. After \( n \) iterations, Matrix \( I(s+n) \) is compressed, using the Run Length Encoding (RLE) method before being sent to the Potential Field Creation procedure.

**Potential Field Creation.** Image matrix \( I(s+n) \) is decompressed and displayed on the user interface. Next, the user interacts with the user interface to define a C-Space, based on the workspace image. Using the computer mouse, the user indicates the obstacles' vertices on the image, and the system calculates and draws a convex
Figure 6.2: Block diagram of the real-time control for Internet-based teleoperation.

Table 6.1: Definition of variables used in the block diagram shown in Figure 6.2.

<table>
<thead>
<tr>
<th>Variable Definition</th>
<th>Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_m(s) = [F_{mx}(s)F_{my}(s)]^T$</td>
<td>Applied force by joystick</td>
</tr>
<tr>
<td>$X_m(s) = [X_{mx}(s)X_{my}(s)]^T$</td>
<td>Joystick position</td>
</tr>
<tr>
<td>$V_d(s) = [V_{dleft}(s)V_{dright}(s)]^T$</td>
<td>Desired velocity</td>
</tr>
<tr>
<td>$V_a(s) = [V_{aleft}(s)V_{aright}(s)]^T$</td>
<td>Applied velocity</td>
</tr>
<tr>
<td>$V_m(s) = [V_{mleft}(s)V_{mright}(s)]^T$</td>
<td>Measured velocity</td>
</tr>
<tr>
<td>$E(s) = [E_x(s)E_y(s)]^T$</td>
<td>Tracking error</td>
</tr>
<tr>
<td>$V_e(s) = [V_{eleft}(s)V_{eright}(s)]^T$</td>
<td>Potential field velocity</td>
</tr>
<tr>
<td>$P_k(s) = [X(s)Y(s)\theta(s)]^T$</td>
<td>Robot position</td>
</tr>
</tbody>
</table>

polygon that is expanded in relation to the robot’s radius to build a Configuration Obstacle $C$-Obstacle. This procedure is carried out $m$ times, once for each physical obstacle. After that, the system converts the displayed $C$-Space bitmap into a numeric matrix $M$ with the same size as $I(s)$, where obstacles are expanded in a decremented way to create a numeric potential field. The numeric matrix $M$ is calculated once.

**Image Processing.** Matrix $I(s)$ is binarized and segmented to find every object in the image. Found objects are characterized, using Hu invariant moments to identify a robot’s artificial landmark. Two circles compose the robot’s landmark: a big circle represents the robot’s front, while a small one indicates the rear. The robot’s position $P_k(s)$ is the centroid of the big circle. The robot’s orientation is the angle defined by a line passing through the centroids of both circles and the $x$-axis of the reference frame.

**Operator and Joystick.** The operator, in conjunction with the joystick, has a spring-like behavior [Nogan 1986]; the operator generates new joystick positions to
Table 6.2: Equations for joystick position conversion into desired velocity.

<table>
<thead>
<tr>
<th>( X_{mx}(s), X_{my}(s) )</th>
<th>( V_{dleft} )</th>
<th>( V_{dright} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+X_{mx}(s), -X_{my}(s))</td>
<td>(-\frac{X_{mx}(s)}{K_p})</td>
<td>(-\frac{X_{my}(s) - X_{mx}(s)}{K_p} )</td>
</tr>
<tr>
<td>(+X_{mx}(s), +X_{my}(s))</td>
<td>(-\frac{X_{my}(s)}{K_p})</td>
<td>(-\frac{X_{my}(s) + X_{mx}(s)}{K_p} )</td>
</tr>
<tr>
<td>(-X_{mx}(s), -X_{my}(s))</td>
<td>(-\frac{X_{my}(s) + X_{mx}(s)}{K_p})</td>
<td>(-\frac{X_{my}(s)}{K_p} )</td>
</tr>
<tr>
<td>(-X_{mx}(s), +X_{my}(s))</td>
<td>(-\frac{X_{my}(s) - X_{mx}(s)}{K_p})</td>
<td>(-\frac{X_{my}(s)}{K_p} )</td>
</tr>
<tr>
<td>(+X_{mx}(s), 0)</td>
<td>0</td>
<td>(-\frac{X_{mx}(s)}{K_p \times K_p})</td>
</tr>
<tr>
<td>( X_{mx}(s), 0)</td>
<td>(-\frac{X_{my}(s)}{K_p \times K_p})</td>
<td>0</td>
</tr>
</tbody>
</table>

compensate for displacements generated by force feedbacks, according to the following function:

\[
X_m(s) = \frac{F_m(s - 1)}{K_m},
\]

where \( K_m \) is a scaling constant, \( X_m(s) \) is the new joystick position, and \( F_m(s - 1) \) is the previously-played force. Notice that the rotational component \( X_{m0}(s) \) is not used, since the joystick is not able to feed force in that direction.

**Joystick Controller.** Joystick position \( X_m(s) \) is proportionally converted to a desired velocity \( V_d(s) \) for both left and right robot motors for differential driving. \( X_m(s) \) and equations in Table 6.2 are used to obtain the velocity \( V_d(s) \). Equations in the four first rows use \( K_x \) to decrease the joystick position component \( X_{mx}(s) \) to generate a hyperbolic speed function on the x-axis of the joystick reference frame. \( K_p \) is a scaling constant, that is, \( K_x = 3 \), \( K_p = 100 \).

**Proximity and Direction Calculation.** Potential field measurements are performed, taking as reference the robot position \( P_k(s) \). These measurements are treated as proximity sensor readings to generate virtual forces, which are converted to velocity values to be fed into the real-time control. A punctual measure over the numeric potential field is called the virtual sensor \( M_0(s) \). Six virtual sensors are distributed around the robot's front (as the real ones) and another six are placed around the robot's rear. These virtual sensors make proximity measures on the numeric potential field grid around obstacles. Proximity measurements are condensed by Equations 6.3 and 6.4 in two variables: proximity \( P_r(s) \) and direction \( \phi(s) \) [Diard and Lebeltel 1999].

\[
P_r(s) = \frac{\text{Max}(M_0(s), M_1(s), M_2(s), M_3(s), M_4(s), M_5(s))}{K_s}
\]

\[
\phi(s) = \frac{90(M_5(s) - M_0(s)) + 45(M_4(s) - M_1(s)) + 5(M_3(s) - M_2(s))}{9(1 + \sum_{i=0}^{5} M_i(s))}
\]

Variable \( P_r(s) \) (proximity) is calculated based on the maximum proximity measure,
Table 6.3: Equations for proximity conversion into potential field velocity.

<table>
<thead>
<tr>
<th>$\phi(s)$</th>
<th>$V_{\text{left}}$</th>
<th>$V_{\text{right}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi(s) = 0$</td>
<td>$P_r(s) \times V_{\text{dleft}}(s)$</td>
<td>$P_r(s) \times V_{\text{dright}}(s)$</td>
</tr>
<tr>
<td>$\phi(s) &gt; 0$</td>
<td>$0$</td>
<td>$K_e P_r(s) \times V_{\text{dright}}(s) \times \cos(10 \phi(s))$</td>
</tr>
<tr>
<td>$\phi(s) &lt; 0$</td>
<td>$P_r \times V_{\text{dleft}}(s) \times \cos(-10 \phi(s))$</td>
<td>$K_e P_r(s) \times V_{\text{dleft}}(s) \times \cos(10 \phi(s))$</td>
</tr>
</tbody>
</table>

which is divided by a scaling constant $K_e$. Variable $\phi(s)$ (direction) is computed, subtracting opposite side measurements of virtual sensors. Therefore, a positive or negative sign is obtained, indicating the left or right side of the robot’s front/rear. Resulting values from the subtraction are multiplied by weights corresponding to virtual sensor positions. The $\phi(s)$ value is obtained by summing the weighted subtraction results. Then, $\phi(s)$ is divided by a summation to normalize the results into a $-10 < 0 < +10$ range. The $P_r(s)$ and $\phi(s)$ behavior is described in detail in [Diard and Lebeltel 1999].

When the robot performs a forward displacement, the six frontal virtual sensors are taken into account to calculate $P_r(s)$ and $\phi(s)$ and generate $V_e(s)$ in accordance with the current robot position $P_k(s)$ in the potential field. When a backward displacement is performed the six rear virtual sensors are taken into account to calculate $P_r(s)$ and $\phi(s)$ and generate $V_e(s)$. The velocity $V_e(s)$ is calculated, based on Table 6.3 equations. In these equations, $P_r(s)$ represents the virtual force vector magnitude and $\phi(s)$ is used to obtain the $P_r(s)$ vector projection on the corresponding axis and to generate the velocity $V_e(s)$ on both left and right motors, using $K_e$ as a scaling constant. After that, the calculated $V_e(s)$ is subtracted from $V_d(s)$ to produce an applied velocity $V_a(s)$ to the robot:

$$V_a(s) = V_d(s) - V_e(s) \quad (6.5)$$

**Conversion to Robot Command.** Velocity $V_a(s)$ is converted into a command string recognized by the robot speed mode [K-Team 2002], with the corresponding parameters set to both left and right wheel motor speeds ($V_{\text{left}}(s), V_{\text{right}}(s)$).

**Slave (mobile robot).** The robot receives the speed command string and executes and holds the command for a specific lapse of time, then stops and waits for the next command. Speed commands set each wheel motor speed through the robot Proportional-Integrative-Differential (PID) controller and a Pulse Width Modulator (PWM) [K-Team 1998].

**Speed Acquisition.** Once the velocity $V_a(s)$ is applied to the robot, the speed acquisition procedure requests the instantaneous speed [K-Team 2002] of both wheel motors ($V_{\text{dleft}}(s), V_{\text{dright}}(s)$) to be sent as the measured velocity $V_m(s)$ to the user computer. $V_d(s)$ is taken as a reference; the difference between $V_d(s)$ and $V_m(s)$ generates a tracking error $E(s)$ that is converted into a force vector $F_m(s)$ by the joystick.
6.2. Real-Time Control

The joystick controller converts error $E(s)$ into force vector $F_m(s)$, applying equations in Table 6.2 in an inverse manner to obtain error projection $E(s) = [E_x(s) E_y(s)]^T$ on the $x$ and $y$ axes. This vector $E(s)$ is added to the last applied force $F_m(s - 1)$. The resulting vector $F'_m(s)$ is multiplied by a scaling constant $K_f$ to obtain the current force vector $F_m(s)$. Then, the $F_m(s)$ vector is played by the joystick during a specific lapse of time.

$$
F'_m(s) = F_m(s - 1) + E(s) 
$$

(6.7)

$$
F_m(s) = K_f F'_m(s) 
$$

(6.8)

**Internet.** This controller design takes the communication link as a delay element that plays no role in the event-based control model [Elhajj et al. 2000b].

### 6.2.2 Experiment Processes

Figure 6.3 shows a detailed process flow chart based on the control design description, with necessary processes that should compose control blocks, shown in Figure 6.2, to achieve their specific tasks in the experiment. Notice that some processes (square blocks) were taken, as modules from the previously-developed VL for mobile robotics, to build the process flow chart.

The User side, as shown in Figure 6.2, contains the Operator and Joystick block that are represented by the User, Force Effector, and Joystick devices in Figure 6.3. The Joystick Controller block is represented by the Force Conversion and Position Conversion processes. The Potential Field Creation block is represented by the User Interface, Vertex Convex Hull, Obstacle Expansion, Visual Display, Image Decompression, and Bitmap to Matrix processes. The Proximity and Direction block is represented by itself. Two procedures were added: the C-Space Database and Position Prediction Calculation Processes.

In the Laboratory side, the Image Acquisition block is represented by the Image Acquisition and Compression procedures. The Image Processing block is represented by several processes: Binarization, Segmentation, Characterization, and Position Calculation. The conversion to Robot Command block is represented by the Control Process. The slave device is represented by the Odometer and Robot Devices. The speed acquisition block and the camera are represented by themselves.

The Internet block is represented by the Communication procedure to transmit and receive data between User and Laboratory Computers.
Figure 6.3: Detailed process flow chart of the teleoperation system. Square blocks represent processes that were taken as software modules from the previously-designed VL.

### 6.2.3 Experiment Functionality Analysis

Once the experiment design is finished, the next step is to perform a functionality analysis, based on the experiment description. The application will be designed to have three main functionalities to perform a remote true experiment [Hicks 1982] in accordance with the VL framework defined in Chapter 3.

Three main functionalities will compose the experiment:

**C-Space Creation:** This functionality encapsulates necessary procedures to transform the obstacle's vertices, marked by the user, into convex polygons to represent physical obstacles. These polygons are expanded, taking into account the robot’s shape to create C-Obstacles and simulate a C-Space, which is displayed on a workspace image obtained from the Laboratory Computer that acquires the image from a camera. Note that a C-Space database is considered. Thus, if a C-Space was saved before, then the user is able to retrieve it. In that case, some procedures of the C-Space Creation functionality can be skipped.

**Potential Field Creation:** Once the C-Space is defined by the user, the generated bitmap used to display the C-Space is converted into an integer matrix to be used as a numerical grid. The matrix is processed by a potential field procedure
to introduce the potential function in Equation 6.1. This procedure generates a
decremented potential field around the obstacles that is known as the Constrained
Space in Figure 6.1.

**Robot Teleoperation:** When the user starts the robot teleoperation, the Laboratory
Computer acquires the workspace image, identifies the robot landmark, and sends
a compressed image and the robot position to the User Computer. This robot
position is used to calculate proximity and direction to obstacles around the robot.
Proximity and Direction are generated from several measurements taken on the
potential field numeric grid. A virtual force vector is generated from proximity
and direction parameters and converted into an environment speed. Then, the
user manipulates the joystick, and its position is converted into a desired speed.
The desired speed is compared with the environment speed to produce an applied
speed. The applied speed is sent to the Laboratory Computer system and finally
to the robot. Robot wheel encoders acquire the instantaneous speed that is
requested by the Laboratory Computer. The instantaneous speed is sent to the
User Computer to compare it with the desired speed. The calculated tracking
error is converted into a force vector that is sent to the force effector (the same
joystick), which provides a force feedback to the user. Meanwhile, the robot
position and the desired speed are introduced into a predictor procedure that
provides future robot positions, which are also displayed as simulated landmarks
on the workspace image.

### 6.3 C-Space Creation Functionality Modeling

The C-Space Creation functionality is identified as necessary procedures to mount the
bilateral teleoperation experiment. The modeling of C-Space Creation functionality,
which includes the first three phases of the VL development methodology, is referenced
to Chapter 5, section 5.2. In the bilateral teleoperation experiment, the C-Space
Creation functionality has added two new procedures for image compression and
decompression, which are taken into account in the modeling phases.

Subsection 5.2.1 described the first phase of the methodology performing an
analysis of the C-Space Creation functionality. The decompression function should be
added to the Visual Display process and the compression function added to the Image
Processing process. Subsection 5.2.2 detailed the second phase of the development
methodology. Notice that the UML class diagram of the C-Space Creation subsystem
in the VL for bilateral teleoperation must take into account two new operation
methods for the Image Processing and Visual Display instantiations: compress() and
deCompress(), respectively.

The second stage of the subsystem identification phase described the dynamics of
the C-Space Creation functionality. The third stage of the identification phase pro-
vides a detailed description of the functionality operation. The image decompression
activity must be added to the transition label that describes the Process and Display operation, and the image compression activity must be added to the transition label that describes the capture and acquire image operation.

The third phase of the VL development methodology converts the UML statechart diagram into a Petri Net. The event-synchronization property was validated via Petri Net behavioral properties. The C-Space Creation subsystem analysis demonstrated its event-synchronicity property and, thus its stability as an event-based design.

6.4 Potential Field Creation Functionality

Modeling

The Potential Field Creation functionality is identified as the necessary procedures to define the robotic navigation experiment to introduce specific independent variables into the experiment. The Potential Field Creation procedures define the decremented potential field around the obstacles, which is used as a reference to generate virtual forces that depend on the obstacle proximity. This degraded potential field can be defined using logarithmic, linear, or exponential equations that change virtual sensor detection ranges and specifications.

This section summarizes the first three methodology phases used to translate the Potential Field Creation functionality into UML and Petri Net modeling notations.

6.4.1 Analysis of the Potential Field Creation Functionality

To continue the first phase of the development methodology, a specification table and a hierarchical chart are build to describe inherent procedures of the Potential Field Creation functionality, specifications, and functions. Table 6.4 extracts and summarizes the most significant processes, including functions and specifications. Notice that physical devices are also described as processes with functions and specifications.

The next step in the experiment analysis generates a functionality hierarchical chart from the specification table. Figure 6.4 describes processes and functions for the Potential Field Creation functionality.

6.4.2 Potential Field Creation Subsystem Identification

Until now, the Potential Field Creation functionality was informally described. In the second phase of the development methodology, experiment processes and func-
Table 6.4: The most significant specifications of the Potential Field subsystem.

<table>
<thead>
<tr>
<th>Process</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>Displays GUI, workspace image, and potential field</td>
</tr>
<tr>
<td>Mouse</td>
<td>Generates discrete signals to start the Potential Field Creation</td>
</tr>
<tr>
<td>Visual Display</td>
<td>Image I decompression and display processing</td>
</tr>
<tr>
<td>Potential Field</td>
<td>Converts Bitmap C into the C-Space matrix $C = {c_{320,240}}$, performs the linear function 6.1 around the obstacles in $C$</td>
</tr>
<tr>
<td>User Interface</td>
<td>Gets start command</td>
</tr>
</tbody>
</table>

Figure 6.4: Hierarchical chart for the Potential Field Creation functionality.

The first stage of the identification phase is structural modeling. Now, the functionality is treated as a subsystem. This stage relates the specification table and hierarchical chart to UML class and deployment diagrams, taking as a reference the VL definition provided by the UML framework. Figure 6.5 shows the class structure and operation methods of the Potential Field Creation functionality.

Figure 6.5 depicts the most significant instantiations of the Guest class in the Potential Field Creation subsystem: Guest Computer, Monitor, and Guest Software. The Guest computer is identified as a computer where the user will interact to create the Potential Field. The PC Mouse is identified as an Input Device instantiation, which is a component of the Guest Computer. The Monitor is identified as an Output Device instantiation in the Guest System. The Guest Software has several components: 1) the Display Process as an instantiation of the Sensory Conversion class with two operation methods: displayImage() and displayCspace(). 2) The User Interface as an
Input Processing instantiation, with getCreateCmnd() as operation method to acquire the creation command from the user. 3) The Potential Field Creation procedures as a Simulation instantiation with two operation methods: bitmapToMatrix() and potentialCalcula(), which are used to create a numerical matrix from the C-Space bitmap and calculate a decremented potential field around obstacles, respectively.

The second stage of the identification phase creates UML collaboration and sequence diagrams from the informal definition to represent the dynamics of the Potential Field Creation subsystem. Instantiation relationships, messages, and sequences are described in detail in this stage. The UML collaboration diagram of the Potential Field Creation subsystem is depicted in Figure 6.6.
The UML collaboration diagram in Figure 6.6 describes the flow of messages among instantiations. The User generates analog Inputs via the Mouse, which translates these Inputs into discrete commands to start the Potential Field Creation from the User Interface procedures. The User interface procedure introduces the Creation Command into the Potential Field Creation instantiation, which defines the C-Space for the Display Process. The Display Process sends a C-Space bitmap to Potential Field procedures, where the bitmap is used to generate a numeric matrix. Furthermore, the Display Process sends the C-Space Image Signal to the Monitor to display the C-Space. This sequence description is clearly represented by the UML sequence diagram, shown in Figure 6.7.

Figure 6.7: UML sequence diagram of the Potential Field Creation subsystem.

A UML statechart diagram is generated in the third stage of the VL framework identification phase to give a detailed description of the VL subsystem operation. This statechart diagram describes operations and internal states performed by the interaction of identified instantiations in the customized VL framework. Identified instantiations are listed and numbered to relate them to internal states and transitions enclosed in dashed boxes in the statechart diagram. Figure 6.8 shows internal states of the Potential Field Creation subsystem. UML statecharts describe internal states of VL subsystems as a result of the last transition (operation).

6.4.3 UML-Petri Net Conversion and Analysis

The UML-Petri Net conversion is the third phase of the methodology; it transforms the UML statechart diagram into a Petri Net using the conversion function $G(ST) = PN$. This conversion generates a Petri Net ($PN$) from the previously shown
Chapter 6. Design of a Virtual Laboratory for Bilateral Teleoperation

Figure 6.8: UML statechart diagram of the Potential Field Creation subsystem.

The dynamic analysis performed on the Potential Field Creation Petri Net, shown in Figure 6.9, describes the Petri Net as a pure and ordinary net [Murata 1989] that behaves as a marking graph that does not have conflicts in its structure; therefore, this Petri Net graph exhibits a deterministic behavior. Its coverability tree shown in Figure 6.10, describes a safe net and all transitions have one input and one output as a state machine, where all transitions are L1-live. Thus, it is an event-synchronized subsystem where its goal state can be reached (p5, potential field matrix created).
6.4. Potential Field Creation Functionality Modeling

Figure 6.9: Petri Net diagram of the Potential Field Creation subsystem.

\[ M = p_1, p_2, p_3, p_4, p_5 \]

\[ M_0 = 1,0,0,0,0 \]

\[ \begin{align*}
M_1 & = 0,1,0,0,0 \\
& \downarrow t_1 \\
M_2 & = 0,0,1,0,0 \\
& \downarrow t_2 \\
M_3 & = 0,0,0,1,0 \\
& \downarrow t_3 \\
M_4 & = 0,0,0,0,1 \\
& \downarrow t_4 \\
\end{align*} \]

Figure 6.10: Coverability tree of the Potential field Creation subsystem.
6.5 Robot Teleoperation Functionality Modeling

Appendix D, section D.2 describes the Robot Teleoperation functionality modeling phases. The functionality modeling describes the first three phases of the VL development methodology: experiment functionality analysis, experiment subsystem identification in the VL framework, and UML statechart conversion into a Petri Net and its analysis.

Descriptions for the bilateral teleoperation experiment are presented in Appendix D, such as, its specification table and hierarchical chart. These descriptions are translated into the UML graphical notation as class, collaboration, sequence, and statechart diagrams of the UML framework. The framework dynamics is extracted (statechart diagram) to be converted into a Petri Net, which is analyzed to relate behavioral properties with event-based control properties, using the coverability tree method. Event-synchronization property is validated via liveness, boundedness, and automaton characteristics of the Petri Nets that represent the Robot Teleoperation functionality.

6.6 Petri Net Synchronous Composition

Analysis results from the third phase of the development methodology allow concluding that these three VL subsystems can accomplish their goals. They meet the event-based design requirements and can perform their specific goals. These VL subsystems are represented by Petri Nets with necessary behavioral properties to assure their own stability and synchronization when they work as separated event-based Internet applications. However, these subsystems should be merged to be considered as a complete VL system, which sequentially performs the experiment functionalities for a true experiment experience.

Transition synchronous composition and place fusion reduction procedures merge VL subsystems into a composed Petri Net. The first synchronous composition and place fusions are done merging $PN_1 || ^+ PN_2 = PN_3$, the C-Space Creation ($PN_1$) and the Potential Field ($PN_2$) subsystems. The composed Petri Net ($PN_3$) is once again synchronously composed and place fused $PN_3 || ^+ PN_4 = PN_5$ with the Robot Teleoperation subsystem ($PN_4$); the resulting composed Petri Net ($PN_5$) is shown in Figure 6.11. The Petri Net numeric labeling starts, using as a reference, the first subsystem to create a new joined labeling. Since merged VL subsystems should be performed in a sequential way, the initial marking of the composed Petri Net is always placed where the first subsystem starts ($p_1$, camera ready). In this case, the initial marking has a token placed at $p_1$ of the C-Space Creation Petri Net. Note that textual labels are not modified because they are taken as a reference to perform the synchronous composition and place fusion procedures.
Figure 6.11: Composed Petri Net diagram resulting from synchronous composition and place fusion of the three VL subsystems: C-Space Creation, Potential Field Creation, and Robot Teleoperation.
Table 6.5: Joined place numeric labeling and its description.

<table>
<thead>
<tr>
<th>Place</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>camera ready to capture</td>
</tr>
<tr>
<td>$p_2$</td>
<td>image acquired</td>
</tr>
<tr>
<td>$p_3$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_4$</td>
<td>image received</td>
</tr>
<tr>
<td>$p_5$</td>
<td>image displayed</td>
</tr>
<tr>
<td>$p_6$</td>
<td>vertices defined</td>
</tr>
<tr>
<td>$p_7$</td>
<td>obstacle defined</td>
</tr>
<tr>
<td>$p_8$</td>
<td>$C$-Obstacle built</td>
</tr>
<tr>
<td>$p_9$</td>
<td>$C$-Obstacle displayed</td>
</tr>
<tr>
<td>$p_{10}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{11}$</td>
<td>$C$-Space defined</td>
</tr>
<tr>
<td>$p_{12}$</td>
<td>potential field creation started</td>
</tr>
<tr>
<td>$p_{13}$</td>
<td>numeric matrix created</td>
</tr>
<tr>
<td>$p_{14}$</td>
<td>potential field matrix created</td>
</tr>
<tr>
<td>$p_{15}$</td>
<td>potential field matrix defined</td>
</tr>
<tr>
<td>$p_{16}$</td>
<td>force played</td>
</tr>
<tr>
<td>$p_{17}$</td>
<td>joystick position generated</td>
</tr>
<tr>
<td>$p_{18}$</td>
<td>desired speed calculated</td>
</tr>
<tr>
<td>$p_{19}$</td>
<td>environment force calculated</td>
</tr>
<tr>
<td>$p_{20}$</td>
<td>environment speed calculated</td>
</tr>
<tr>
<td>$p_{21}$</td>
<td>applied speed calculated</td>
</tr>
<tr>
<td>$p_{22}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{23}$</td>
<td>speed command received</td>
</tr>
<tr>
<td>$p_{24}$</td>
<td>robot moving</td>
</tr>
<tr>
<td>$p_{25}$</td>
<td>speed measured</td>
</tr>
<tr>
<td>$p_{26}$</td>
<td>image binarized</td>
</tr>
<tr>
<td>$p_{27}$</td>
<td>object array filled</td>
</tr>
<tr>
<td>$p_{28}$</td>
<td>robot landmark recognized</td>
</tr>
<tr>
<td>$p_{29}$</td>
<td>robot position calculated</td>
</tr>
<tr>
<td>$p_{30}$</td>
<td>buffer full</td>
</tr>
<tr>
<td>$p_{31}$</td>
<td>image, position, and speed received</td>
</tr>
<tr>
<td>$p_{32}$</td>
<td>predicted position calculated</td>
</tr>
<tr>
<td>$p_{33}$</td>
<td>image and predicted position displayed</td>
</tr>
<tr>
<td>$p_{34}$</td>
<td>speed error calculated</td>
</tr>
<tr>
<td>$p_{35}$</td>
<td>force vector calculated</td>
</tr>
<tr>
<td>Transition</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>$t_1$</td>
<td>capture and acquire (compress image)</td>
</tr>
<tr>
<td>$t_2$</td>
<td>transmit image</td>
</tr>
<tr>
<td>$t_3$</td>
<td>receive image</td>
</tr>
<tr>
<td>$t_4$</td>
<td>process (decompress) and display image</td>
</tr>
<tr>
<td>$t_5$</td>
<td>get vertices</td>
</tr>
<tr>
<td>$t_6$</td>
<td>calculate convex hull</td>
</tr>
<tr>
<td>$t_7$</td>
<td>expand convex hull</td>
</tr>
<tr>
<td>$t_8$</td>
<td>process and display $C$-Space</td>
</tr>
<tr>
<td>$t_9$</td>
<td>transmit image request</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>receive image request</td>
</tr>
<tr>
<td>$t_{11}$</td>
<td>get acceptance of $C$-Space</td>
</tr>
<tr>
<td>$t_{12}$</td>
<td>write $C$-Space</td>
</tr>
<tr>
<td>$t_{13}$</td>
<td>read obstacle definition</td>
</tr>
<tr>
<td>$t_{14}$</td>
<td>get create command</td>
</tr>
<tr>
<td>$t_{15}$</td>
<td>convert bitmap into matrix</td>
</tr>
<tr>
<td>$t_{16}$</td>
<td>create potential field</td>
</tr>
<tr>
<td>$t_{17}$</td>
<td>process and display potential field</td>
</tr>
<tr>
<td>$t_{18}$</td>
<td>get start</td>
</tr>
<tr>
<td>$t_{19}$</td>
<td>get joystick position</td>
</tr>
<tr>
<td>$t_{20}$</td>
<td>calculate desired speed</td>
</tr>
<tr>
<td>$t_{21}$</td>
<td>calculate environment force</td>
</tr>
<tr>
<td>$t_{22}$</td>
<td>convert environment force into environment speed</td>
</tr>
<tr>
<td>$t_{23}$</td>
<td>subtract environment speed to desired speed</td>
</tr>
<tr>
<td>$t_{24}$</td>
<td>transmit applied speed</td>
</tr>
<tr>
<td>$t_{25}$</td>
<td>receive applied speed</td>
</tr>
<tr>
<td>$t_{26}$</td>
<td>execute command by robot</td>
</tr>
<tr>
<td>$t_{27}$</td>
<td>acquire speed by odometer</td>
</tr>
<tr>
<td>$t_{28}$</td>
<td>execute stop command by robot</td>
</tr>
<tr>
<td>$t_{29}$</td>
<td>perform binarization</td>
</tr>
<tr>
<td>$t_{30}$</td>
<td>perform segmentation</td>
</tr>
<tr>
<td>$t_{31}$</td>
<td>perform characterization</td>
</tr>
<tr>
<td>$t_{32}$</td>
<td>calculate robot position</td>
</tr>
<tr>
<td>$t_{33}$</td>
<td>transmit image, position, and speed</td>
</tr>
<tr>
<td>$t_{34}$</td>
<td>receive image, position, and speed</td>
</tr>
<tr>
<td>$t_{35}$</td>
<td>calculate predicted position</td>
</tr>
<tr>
<td>$t_{36}$</td>
<td>process (decompress), display image, and predicted position</td>
</tr>
<tr>
<td>$t_{37}$</td>
<td>subtract measured speed to desired speed</td>
</tr>
<tr>
<td>$t_{38}$</td>
<td>convert error to force vector</td>
</tr>
<tr>
<td>$t_{39}$</td>
<td>play force</td>
</tr>
</tbody>
</table>
6.6.1 Composed Petri Net Analysis

The composed Petri Net shown in Figure 6.11 inherits behavioral and structural properties from merged Petri Nets. It has non-deterministic, boundeness (safe), and liveness properties. But the merge increased the non-deterministic property in the complete system. Therefore, the composed Petri Net does not behave as it should to accomplish the goals of VL subsystems.

From the coverability tree analysis results shown in Figure 6.12, it can be deduced that it is necessary to disable part of the composed Petri Net. In this case, the C-Space Creation subsystem must be disabled to avoid redefining the C-Space when the system is performing the robot teleoperation functionality. This condition is described by the following statement: if $t_{11}$ (C-Space defined) is fired and $\mu(p_{11}) = 1$, then the C-Space Creation places ($\{p_3, p_4, p_5, p_6, p_7, p_8, p_9\}$) must be permanently disabled. Thus, these places should not have any token after $t_{11}$ was fired.

Furthermore, when the C-Space Creation subsystem is working, the robot teleoperation subsystem $\{p_{26}, p_{27}, ..., p_{35}\}$ must be disabled. This condition is described by the following statement: if $t_{11} = 0$ (C-Space defined) and $\mu(p_{11}) = 0$, then $t_{29} = 0$.

Figure 6.12: Coverability tree of the synchronous composed Petri Net.
6.7 Petri Net Control

The composed Petri Net is then complemented with a non-maximally permissible controller net that meets the following constraint:

\[ \mu(p_{11}) \neq 0, \quad \text{then} \quad \mu(p_3) + \mu(p_4) + \mu(p_5) + \mu(p_6) + \mu(p_7) + \mu(p_8) + \mu(p_9) = 0. \quad (6.9) \]

Then, the controller net is synthesized using the proposed control method.

1. First, the sets \( L_c, L_c^o, L_c^e, CL_{ct}, L_{pure-ct}^o \), and \( L_{pure-ct}^o \) are evaluated.

\[
\begin{align*}
L_c &= \{p_3, p_4, p_5, p_6, p_7, p_8, p_9\} \quad (6.10) \\
L_c^o &= \{t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\} \quad (6.11) \\
L_c^e &= \{t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{11}\} \quad (6.12) \\
CL_{ct} &= \{t_3, t_4, t_5, t_6, t_7, t_8, t_9\} \quad (6.13) \\
L_{pure-ct}^o &= \{t_2\} \quad (6.14) \\
L_{pure-ct}^e &= \{t_{11}\} \quad (6.15)
\end{align*}
\]

2. Construct a controller place \( p_c \).

3. Draw an arc between place \( p_c \) and transition \( t_2 \). Let \( p_c \) be the output of \( t_2 \).

4. Draw an inhibitor arc between \( p_c \) and \( t_{11} \). Let \( t_{11} \) be the input of the inhibitor arc. Place \( p_c \) is named as \( p_{37} \) in the controlled Petri Net, as shown in Figure 6.13. Notice that \( p_{37} \) disables transition \( t_2 \) corresponding to a procedure in the Host system; therefore, a net representing communication between Guest and Host is needed. Two transitions representing transmission and reception operations (\( t_{41} \) and \( t_{42} \)), with a middle buffer full state (\( p_{38} \)), are incorporated, as shown in Figure 6.13. Place \( p_{39} \) indicates that the disable signal was received, having as output arc the inhibitor connected to \( t_2 \).

The second constraint is directly implemented. Since the constraint depends on the firing of \( t_{11} \), which is in the Guest system, and \( t_{29} \) which is in the Host system, then it uses part of the controller net synthesized for the first constraint to avoid sending another signal through the Media. Therefore, it is introduced in another place (\( p_{40} \), enable teleoperation); \( p_{40} \) is an output place of \( t_{42} \) (receive disable) and it is connected as the input place to \( t_{29} \). Once the Robot Teleoperation subsystem is enabled, it must remain enabled, thus another arc from \( t_{29} \) to \( p_{40} \) is introduced into the controller net to generate a self-loop in \( p_{40} \).

The composed Petri Net with the controller net is shown in Figure 6.13. The composed and controlled Petri Net is analyzed once again to verify that experiment specifications are met.
Figure 6.13: Petri Net diagram of the Teleoperation system with controller net structures.
6.7. Petri Net Control

6.7.1 Final Dynamic Analysis

The coverability tree of the composed and controlled Petri Net of the VL for Teleoperation is shown in Figure 6.14. It was done with four added places and two transitions of controller net structures, which are described in Tables 6.7 and 6.8.

Table 6.7: Places added by controller net structures in the VL for teleoperation.

<table>
<thead>
<tr>
<th>Place</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p_{37}</td>
<td>disable C-Space Creation</td>
</tr>
<tr>
<td>p_{38}</td>
<td>buffer full</td>
</tr>
<tr>
<td>p_{39}</td>
<td>disable received</td>
</tr>
<tr>
<td>p_{40}</td>
<td>enable teleoperation</td>
</tr>
</tbody>
</table>

Table 6.8: Transitions added by controller net Structures in the VL for teleoperation.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_{41}</td>
<td>transmit disable</td>
</tr>
<tr>
<td>t_{42}</td>
<td>receive disable</td>
</tr>
</tbody>
</table>

From the coverability tree analysis shown in Figure 6.14, it can be deduced that each VL subsystem reaches its goal state in a sequenced and synchronized way. This means that there exists three firable transition sequences \( \{\tau_1, \tau_2, \tau_3\} \) that do not consider the controller net transitions \( \{t_{41}, t_{42}\} \), that is:

\[
\tau_1 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\} \quad (6.16)
\]

\[
\tau_2 = \{t_{11}, t_{14}, t_{15}, t_{16}, t_{17}\} \quad (6.17)
\]

\[
\tau_3 = \{t_{18}, t_{19}, t_{20}, t_{21}, t_{22}, t_{23}, t_{24}, t_{25}, t_{26}, t_{27}, t_{28}, t_1, t_{29}, t_{30}, t_{31}, t_{32},
    t_{33}, t_{34}, t_{35}, t_{36}, t_{37}, t_{38}, t_{39}, t_{19}, t_{20}, t_{21}, t_{22}, t_{23}, t_{24}, t_{25}, t_{40}\} \quad (6.18)
\]

and their concatenation \( \tau = \tau_1 + \tau_2 + \tau_3 \) allows:

\[
M_0[\tau > M_3 \Rightarrow M_0[\tau_1] > M_1 \text{ and } M_1[\tau_2] > M_2 \text{ and } M_2[\tau_3] > M_3 \text{ are defined,} \quad (6.19)
\]

where \( M_1 \) represents the marking vector of the C-Space Creation subsystem that includes \( \mu(p_{11}) = 1 \) (C-Space defined), considering that other place markings of the composed net are equal to zero \( \{\mu(p_1) = 0, \ldots, \mu(p_{36}) = 0\} \) and taking no care regarding controller net place markings \( \{\mu(p_{37}), \mu(p_{38}), \mu(p_{39}), \mu(p_{40})\} \). \( M_2 \) represents the marking vector of the Potential Field Creation subsystem, which includes \( \mu(p_{15}) = 1 \) (potential field matrix defined), taking into account the previously mentioned marking considerations. \( M_3 \) is the marking of the Robot Teleoperation subsystem that includes
$\mu(p_{36}) = 1$ (destination reached), taking into account the marking considerations. Notice that $p_{36}$ is the VL goal state.

Without taking into account the marking of the controller net places \{$p_{37}$, $p_{38}$, $p_{39}$, $p_{40}$\} and controller net transitions \{$t_{11}$, $t_{22}$\}, the composed Petri Net behaves as a safe net (1-bounded), where all its transitions are firables and have one input and output place, as a state machine automaton. Thus, it means there exists only one signal instantiation at any instant in the VL system.

Furthermore, $t_{40}$ (stop robot) is L1-live, and according to Eq. 6.19, $p_{36}$ is reachable from $M_0$; this implies that the VL goal state is also reachable. Therefore, the composed Petri Net is event-synchronized and performs sequentially and correctly its functionalities to mount, define, and execute the experiment.
6.8 VL System Implementation

The composed and controlled Petri Net still has appropriate boundeness, liveness, and event-synchronous properties. Therefore, it is a stable event-based design for the Internet [Elhajj et al. 2003].

The VL system for bilateral teleoperation over the Internet can be implemented using UML diagrams to build up its software structure in object-oriented programming languages. The composed and controlled Petri Net diagram tells the developer and programmers how messages among instantiations must be implemented to sequentially perform the C-Space Creation, Potential Field definition, and Robot Teleoperation functionalities in the telerobotic experiment.

6.9 VL Development Conclusions

This chapter described the structural and dynamic design of a VL for bilateral teleoperation using the VL development methodology. This development took advantage of existing software classes, developed in the previous chapter, to create a new experiment design for Internet-based teleoperation with haptic and visual feedback.

The experiment description introduced a real-time, event-based control method to teleoperate a nonholonomic minirobot that combines computer vision and potential field techniques, providing visual feedback and force reflection that corresponds to a robot status in the workspace.

In the first methodology phase, the experiment description and its process flow chart were analyzed to generate specification tables and hierarchical charts for experiment functionalities. From this analysis, three main functionalities were recognized to mount, define, and execute the experiment for bilateral teleoperation: C-Space Creation, Potential Field Creation and Robot Teleoperation.

These functionalities were identified in the UML reference framework in the second phase. In this phase, experiment functionalities were treated as VL subsystems and related to the VL framework. Notice that software reuse was done when the C-Space Creation functionality design was taken from the previously developed VL for mobile robotics. Moreover, some existing classes were taken as is, using class encapsulation, or their design was modified to create a new class extension, which includes new operation methods to enhance their capabilities, using class inheritance.

In the third phase, the VL subsystem dynamics was extracted, converting state-chart diagrams into Petri Nets, and performed a quantitative and qualitative analysis to validate Petri Net behavioral properties that are related with event-based design
requirements.

These Petri Nets were merged into a composed Petri Net in the fourth phase. After that, the composed Petri Net was analyzed to verify that experiment functionalities are sequentially executed. Since the composed Petri Nets did not behave as desired, a controller net was synthesized to synchronize the sequential execution of VL subsystems.

The customized framework and its composed Petri Net were taken as a guideline to implement the VL application using Java and C++ programming languages. UML diagrams helped to visualize the software structure and provided a design compatible with the object-oriented programming paradigm. This paradigm allows reuse and modification of previously designed software classes to reduce development efforts and time. Petri Nets provided a fully compatible event-driven programming control through event-listener class interfaces, class messages, and public data in Java and C++ languages.

A detailed description of implemented methods and techniques in the VL system are presented in Appendix B and C. The VL setup and experimental results will be described in Chapter 7.
Chapter 7

Implementation and Experimental Results

This chapter presents implementation details of previously designed and modeled VLs along with their experimental results. Both the VL for mobile robotics and the VL for bilateral teleoperation share common hardware, software, and specifications.

The first section discusses the hardware used to implement both VLs and a complementary Observer system. It includes experimental functionalities, characteristics, and specifications of these applications. Description and specification of implemented software are presented in the second section. The third section describes the geographical location of both Guest and Host systems when remote experimentation was performed, and it includes Internet link specifications. The fourth section provides several specifications regarding the VL for mobile robotics and discusses experimental results, which are analyzed for performance benchmarks described in [Borstel et al. 2003b]. The fifth section describes several specifications of the implemented VL for bilateral teleoperation, and deals with experimental results to analyze accuracy and performance of the VL system. These specifications and experimental results were discussed in [Borstel and Gordillo 2004a]. Concluding remarks regarding experimental results and their analysis are described in the last section.

7.1 Hardware Description and Specifications

The diversity of hardware used to perform remote experiments forced a detailed study of integration and interfacing. The following list describes general hardware used to implement VL applications:

*Guest Computer.* This computer contains user interfaces to perform remote telerobotic experimentation over the Internet. It contains Guest software and devices defined by the customized VL framework and Petri Net diagrams, and is connected to the Internet via an Ethernet board and to output and input devices, such as: monitor, mouse, and joystick.
Chapter 7. Implementation and Experimental Results

*Host Computer.* This computer containing *Host* software and devices defined by the customized VL framework and Petri Net diagrams. It is connected to the Internet, sensors, and mechanism and control devices, such as the *Host* camera and the mobile robot.

*Host Camera.* A visual sensor connected to the *Host* Computer as is defined by the UML deployment diagram. A Sunvideo Plus board is used to capture camera output.

Haptic Device. An input/output device connected to the *Guest* Computer as defined by the UML deployment diagram. It is a programmable force feedback joystick able to obtain commands from the user and render force. The *Guest's* USB port is used to connect the joystick.

Mobile Robot. This is a differential-driven nonholonomic mobile minirobot. It is used as a mechanism and control device connected to the *Host* Computer via its RS232 port. It accepts string commands to control its PID-PWM wheel controller and obtain sensor readings from its integrated infra-red proximeters and wheel encoders.

Observer Computer. This computer is used to implement a separated observer system to provide an enhanced visual feedback to the user through the Internet. The Observer system allows the user to view detailed images of the experiment development.

Observer Camera. This visual sensor is connected to the Observer computer. The Sunvideo Plus board is used to capture camera outputs. The Observer Camera is able to perform pan, tilt, and zoom movements, which are controlled by the Observer Computer via its RS232 port.

Table 7.1 shows hardware component specifications, such as: computer architectures and processors, operating systems, memory, video capture boards, network boards, mobile robot, cameras, and haptic device.

Remote experiments were performed using both *Guest* and *Host* systems in conjunction with the Observer system. The Observer system was implemented to provide an enhanced visual feedback from another point of view of the performed experiment.

7.2 Software Description and Specifications

This section covers general software modeled by the UML and Petri Net designs for both VLS and the Observer system. Software implementation for *Guest* and *Host* systems were almost all written in Java, since this programming language is compatible with the object-oriented paradigm. Sunvideo software drivers were written in C using Sun eXtended Image Libraries (XIL) and integrated with the Java application using
Table 7.1: General hardware description and specifications.

<table>
<thead>
<tr>
<th>Device</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Guest Computer</strong></td>
<td>Compaq computer laptop with a Pentium III processor 1.1GHz, 128 MB in RAM, Windows 2000 operating system</td>
</tr>
<tr>
<td><strong>Host Computer</strong></td>
<td>Sun Workstation Ultra 10, UltraSparc-Ili 333 MHz processor, 320 MB in RAM, V5.7 Sun operating system, equipped with a Sunvideo Plus PCI board with two RCA inputs for NTSC video, and an integrated network Ethernet board 10/100 baseT</td>
</tr>
<tr>
<td><strong>Host Camera</strong></td>
<td>Sun Microsystems video camera, NTSC video output</td>
</tr>
<tr>
<td><strong>Haptic Device</strong></td>
<td>Microsoft Joystick SideWinder Force Feedback Pro, 16-bit coprocessor at 25 Mhz, with infra-red optic position sensors, USB port, 3 degrees of freedom (DOF) position, 2 DOF force rendering, programmable using Microsoft DirectX library</td>
</tr>
<tr>
<td><strong>Mobile Robot</strong></td>
<td>Khepera minirobot, Motorola 68331 16 Mhz microprocessor, 256 Kb in RAM, equipped with a RS232 port, 2 d.c. brushed servo motors with incremental encoders (12 pulses per mm), 8 infra-red proximity and ambient light sensors with up to 50 mm range, robot diameter: 53 mm, Height: 30 mm, and weight approx. 70 g.</td>
</tr>
<tr>
<td><strong>Observer Computer</strong></td>
<td>Sun Workstation Sparc 5, UltraSparc-Ili 270 Mhz processor, 256 MB in RAM, V5.7 Sun operating system, equipped with a Sunvideo Plus PCI board with two RCA inputs for NTSC video, and an integrated network Ethernet board 10/100 baseT</td>
</tr>
<tr>
<td><strong>Observer Camera</strong></td>
<td>Sony EVI-D30 mobile camera with pan, tilt, and zoom, NTSC video output, with a RS232 port for VISCA commands.</td>
</tr>
</tbody>
</table>

Java Native Interface (JNI) methods.

A software application was written in Visual C++ using Microsoft DirectX libraries to interface the joystick input/output discrete signals with the VL for bilateral teleoperation system. The VL for bilateral teleoperation used a TCP connection to receive and transmit data between the VL system and a joystick interfacing application. The implemented Internet-based communication platform for both VLs was TCP because of its reliability and synchronization. TCP had to be used to guarantee the arrival of packets since TCP handles retransmission of lost packets. Thus, the developer does not need to implement lost-packet retransmission in the VL application. The Guest system behaves as client, requesting data and sending commands to the Host system. The Host system behaves as a server, receiving and filling Guest's requirements. Until now, the connection is peer to peer using a scheduling strategy to perform remote experiments.
The Observer system works in an open-loop control mode and was almost all written in Java. It uses a Sunvideo software driver written in C that captures color images from the camera, and compresses color images using JPEG (Joint Photography Expert Group) format. This driver was also integrated with Java, using the JNI method. Two independent client-server applications were implemented for visual feedback and camera control. The communication platform for visual feedback was implemented using Java Remote Method Invocation (RMI). The communication platform for camera control was TCP. This system is explained and described in Appendix E.

### 7.3 Guest and Host Geographic Location and Media Specifications

This section describes the geographic location of Guest and Host systems used to perform remote experiments and the Internet link setup used to transmit control and sensory signals between them.

Experimental results were obtained by planning and executing remotely the teleoperation tasks. The Guest computer was geographically separated from the Host and Observer computers. Guest was located at CIBNOR (Centro de Investigaciones Biológicas del Noroeste La Paz, México) 1,050 km away from the Host system, which was situated in the Robotics and Vision Laboratory of the CSI (Centro de Sistemas Inteligentes, ITESM campus Monterrey, México), as shown in Figure 7.1.

Experimental data obtained from the VL for mobile robotics was acquired using an Internet route with 23 jumps, according to the Unix traceroute analysis. This route had variable bandwidth and large and unpredictable delays. The Host and Observer computers were connected to the ITESM Monterrey campus local network using an Ethernet 100baseT port of a network switch. This local network is connected to the Internet backbone via four E1 links. Each E1 link has a bandwidth of 2,048 Kbps [Group 1996]. The Guest computer was connected to the CIBNOR local network using an Ethernet 100baseT port of a network switch. The CIBNOR local network is connected to the Internet backbone via an E1 link.

Experimental data obtained from the VL for bilateral teleoperation was acquired using an Internet2 connection with 10 jumps in the route, according to the Unix traceroute analysis. The Host and Observer computers were connected using an Ethernet 100baseT network switch to the ITESM local network of its Monterrey campus, which is connected to the Internet2 backbone via an E3 link. An E3 link has a bandwidth of 155 Mbps [Group 1996]. CIBNOR local network is connected to the Internet2 backbone via an E1 link.
7.4 Remote Experimentation Using the VL for Mobile Robotics

This section describes the VL for mobile robotics, its settings, and specifications during the remote experiments. Several improvements were done to enhance security, autonomy, and user interaction of the VL for mobile robotics. These are described in Appendix F. However, these improvements did not impact experimental results, except by the transmitted image reduction explained in the following subsection. Other improvements are not included in this document so as to maintain UML and Petri Net models with a reasonable and understandable complexity.

The VL for mobile robotics was used in conjunction with the Observer system. All user Interfaces were run in the Guest computer and server applications were executed in the Host and Observer computers, as shown in Figure 7.2.

7.4.1 System Specifications

Uncompressed images sent by the Host to the Guest system were 76.8 Kbytes long. The robot position calculated by the robot position calculation process had 6 bytes of data. Planned paths sent to Host usually took less than 320 bytes and were required once during each experiment. Therefore, image data required a large bandwidth while robot position and planned path data did not require it.

The Guest system was originally designed to request an image per iteration to the Host system. Based on experimental results this behavior decreased the VL
Figure 7.2: The Guest system was located at the user site at CIBNOR and the Host and Observer systems were located at the remote laboratory facility at CSI-ITESM.

performance because of image transmission delays over the Internet. Thus, the VL was modified to enhance its performance; a simple method was used to reduce large image delays:

When the Guest system starts, it performs 10 image requests and calculates their average time delay. The Guest system divides the average image delay by the average Host execution time (131.145ms). This action gives \( n \) Host cycles that can be performed in an image delay interval (while the image travels through the communication link). Then, the Guest system starting its eleventh cycle sends a 2-byte length command. The Host reads this command and performs the path following task without sending any response to Guest. Robot movements are held in a specific time frame (60 ms) to avoid system instability, introduced by unpredictable delays, in accordance with the event-based control theory. This is done \( n - 1 \) additional times, then the Guest system sends an image request and the 2-byte command cycle starts again. A detailed description of this image reduction method is presented in Appendix G.

In the Host system, robot serial communication was set at 9600 bauds. Robot commands were speed commands [K-Team 1998], which set each wheel motor speed through a PID controller and a PWM (Pulse Width Modulator). Translation commands were done using +13 pulses/10 ms settings. Orientation commands were done using different ±4 pulses/10 ms settings on each wheel motor. These settings gave an average forward speed of 2.3383 cm/s on straight paths. The angular threshold \( \delta \), that is used to decide which control command is sent to the robot, was set to 7 degrees in the point to point Control Process of the Host system described in Appendix B.
Figure 7.3 shows the Guest interface that has drawn the C-Space used in the remote experiments.

![Diagram of Guest interface](image)

Figure 7.3: The Guest system shows a graphic interface with control buttons and the robot workspace image.

### 7.4.2 Path Analysis

Several path planning and following tasks were performed from the Guest system to analyze the Host system’s performance. Experimental results were calculated, based on a workspace of 73.5 cm width and 98 cm length; this workspace was captured in a 320 x 240 pixels image, which gave a 0.30625 cm/pixel resolution.

Table 7.2 shows results of path analysis for each followed trajectory. Path control points give the amount of points to define robot trajectories. Path distances are the sum of distances between control points. Quadratic error sum and average error calculation are calculated, based on the perpendicular distance among sensed points and their belonging path segment. Quadratic errors represent the area difference between both planned and followed paths.

From Table 7.2, it can be deduced that path errors basically depend on path complexity, and that they were proportional to how many curves exist in planned paths and how many control (clamp) points were used to describe each curve. Straight trajectories can be followed with minimal errors, even if few control points describe them.

Furthermore, the serial cable used to connect the robot to the Host computer, also generates path errors when longer trajectories were followed. This behavior, presented in Figure 7.4, shows the planned path and sensed robot trajectory of the sixth path following task described in Table 7.2. The error area at the path end increased because of the cable’s friction with the workspace surface.
Table 7.2: Path analysis results.

<table>
<thead>
<tr>
<th>Path</th>
<th>Control Points</th>
<th>Total Distance (cm)</th>
<th>Quadratic Error Sum (cm²)</th>
<th>Avg. Error (cm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>49.160</td>
<td>18.536</td>
<td>0.219</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>44.599</td>
<td>20.011</td>
<td>0.243</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>66.092</td>
<td>47.208</td>
<td>0.300</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>30.588</td>
<td>9.670</td>
<td>0.195</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>25.492</td>
<td>5.009</td>
<td>0.060</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>47.063</td>
<td>12.631</td>
<td>0.191</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
<td>97.360</td>
<td>56.955</td>
<td>0.299</td>
</tr>
</tbody>
</table>

Figure 7.4: Planned path and sensed robot trajectory of the sixth experiment (Path 6 in Table 7.2). Chart resolution is 10 pixels per square.

7.4.3 Time Analysis

Time analysis performed on acquired data gave the results shown in Table 7.3. For example, in Path 1, 132.994 seconds were needed to accomplish the path-following, the task, and an uncompressed image sent from the Host system after 37 2-byte commands received from the Guest system; in total, 12 images were sent to the Guest system. The path-follower procedure was executed 478 times in the Host system. Each image added an average time delay of 4.744 seconds.

Average time delay added per image shows longer and unpredictable communication time delays, as shown in Figure 7.5. These long delays impact path-following task time and the number of images sent to the Guest system.

On the other hand, the Host system iteration frequency for each followed path is represented by Host cycles presented in Table 7.3. From the number of Host cycles, it
can be deduced that the number of iterations or cycles of the Host system indicates that the Host frequency is greater than the overall VL system frequency because of the image reduction method.

However, both the Host and overall system iterations are referenced to the robot movements. This is compatible with the event-based control theory where system signals must be referenced to the same event.

<table>
<thead>
<tr>
<th>Path</th>
<th>Total Time (s)</th>
<th>Images Sent</th>
<th>2-byte Cmnds. Sent</th>
<th>Host Cycles</th>
<th>Avg. Delay per Image (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>132.994</td>
<td>12</td>
<td>37</td>
<td>478</td>
<td>4.744</td>
</tr>
<tr>
<td>2</td>
<td>402.734</td>
<td>25</td>
<td>37</td>
<td>951</td>
<td>9.210</td>
</tr>
<tr>
<td>3</td>
<td>603.861</td>
<td>50</td>
<td>37</td>
<td>1899</td>
<td>5.864</td>
</tr>
<tr>
<td>4</td>
<td>94.842</td>
<td>12</td>
<td>26</td>
<td>322</td>
<td>3.752</td>
</tr>
<tr>
<td>5</td>
<td>68.638</td>
<td>10</td>
<td>26</td>
<td>252</td>
<td>3.185</td>
</tr>
<tr>
<td>6</td>
<td>130.988</td>
<td>18</td>
<td>26</td>
<td>494</td>
<td>3.068</td>
</tr>
<tr>
<td>7</td>
<td>386.310</td>
<td>56</td>
<td>27</td>
<td>1439</td>
<td>2.927</td>
</tr>
</tbody>
</table>

Figure 7.5: Time delays of Path 1 in Table 7.3. Plot peaks represent time delays caused by sending an image.
7.5 Experimentation Using the VL for Bilateral Teleoperation

This section describes the experimental setup of the VL for bilateral teleoperation over the Internet, which was complemented with the independent Observer system. All user interfaces were run in the Guest computer, and server applications were executed in the Host and Observer computers. Figure 7.6 shows the experimental configuration.

Figure 7.6: Configuration of the VL for bilateral teleoperation to navigate a small workspace.

7.5.1 System Specifications

The robot workspace and the Guest and Host computers had the same hardware deployment used in remote experiments with the VL for Mobile Robotics. However, several settings were changed in the VL for bilateral teleoperation, such as: the robot motor speeds ranged from ±10 pulses/10 ms and were held for 90 ms to assure system stability over the Internet. For a detailed characterization of the robot PID speed controller and its stability analysis, see Appendix C section C.2.

Figure 7.7 shows teleoperation interfaces. A general user interface, with operation mode buttons at the top, shows the workspace image that has drawn the C-Space used in the experiment and the simulated robot mark that represents the robot’s current position. This simulated landmark is a triangle showing the robot’s position and orientation. The triangle is drawn inside a dotted circle. These dots represent virtual sensors around the robot and indicate their position. A dotted line is drawn
indicating the predicted position of the robot based on the last applied velocity. The user interface on the bottom left side of Figure 7.7 has control buttons used to define the C-Space, build the numerical potential field grid, and start robot teleoperation. The user interface on the bottom right side shows information regarding joystick status and force rendered.

Figure 7.7: User interfaces shown in the Guest Computer of the VL for bilateral teleoperation.

The number of compressed images sent by the Host computer is reduced to avoid unnecessary delays introduced by the image transmission through the Internet. An image is sent to the Guest system every \( n \) events (operator commands). Consequently, after every \( s + n \) event, an image is transmitted and displayed. Transmitted and compressed images ranged from 16 to 19 Kbytes. A detailed description regarding the dynamic design used to reduce transmitted images is presented in Appendix G.

7.5.2 Experimental Results

This subsection describes one of several experiments using the VL for bilateral teleoperation over the Internet2. Figure 7.8 shows the robot trajectory on the \((x,y)\) plane during the experiment; which is 94.710 cm long. The potential field around obstacles is represented by boundary lines and the virtual sensor positions during the robot trajectory are indicated by scattered points. The trajectory plot has a resolution of 0.30625 cm per point. The robot trajectory was performed in 163.445 sec. The average time
for a complete system cycle was 0.9080 sec.

Figure 7.8: Robot trajectory on the \((x, y)\) plane; virtual sensor positions during the trajectory are indicated by scattered points.

Figure 7.9 presents the most significant plots to describe experiment behavior; these plots are referenced to event \(s\); 180 events (commands) were performed. The first row presents the desired velocity \(V_d(s)\) in pulses/10 ms versus \(s\), for both left and right robot motors. The second row shows the sum of measured velocities \(V_m(s)\) and the corresponding \(y\)-axis force \(F_{my}(s)\) versus \(s\). It is clear that these plots are similar, considering that the sum of measured velocities and the force felt in the \(y\)-axis should behave as the desired velocity \(V_d(s)\). The third row illustrates the force rendered by the joystick in both \(x\) and \(y\) axes.

The desired velocity \(V_d(s)\) is decremented according to \(P_r(s)\) and \(\phi(s)\), which are computed using potential field measures acquired by virtual sensors on matrix \(M\), and generate an applied speed \(V_a(s)\) that is sent to the robot. The force feedback \(F_m(s)\) is proportional to the speed tracking error \(E(s)\) that is obtained by subtracting the measured speed \(V_m(s)\) from the \(V_d(s)\). Therefore, if an obstacle is near the robot’s front, then both left and right motor velocities decrease, an inverse force vector increases in the \(y\)-axis of the joystick, and the robot’s rear has the same behavior. If an obstacle is near the right side of the robot’s front, then the right motor velocity decreases. However, a negative force in the \(x\) and \(y\) axes of the joystick increases; this shows that an obstacle is close to that direction. As for the left side, a negative force in the \(y\)-axis and a positive force in the \(x\)-axis is generated. For the robot’s rear, the system behaves in the opposite way. Thus, a closer obstacle causes the tracking error to increase, and this error is proportionally converted to a force vector by the joystick to tell the user that an obstacle is present in the robot trajectory.
7.5. Experimentation Using the VL for Bilateral Teleoperation

Figure 7.9: System behavior during a remote experiment. The first two rows show a comparison among desired velocities and the sum of measured velocities and forces felt. The last row shows the $x$ and $y$ force components. These plots are referenced to the event $s$; 180 events (commands) were performed.

If the robot is located at 3.675 cm from the obstacle, then both left and right motors stop to avoid collision. When virtual sensors have not detected any obstacle around the robot, measured velocity starts tracking the desired one, and both tracking error and force vector decrease. Notice that the tracking error variation is influenced by the tracking error of the robot’s PID controller $\pm 1$ pulse/10 ms, analyzed in Appendix C section C.2, by the friction of robot wheels on the wood-made workspace surface, and
by the friction of the serial cable attached to the robot on the wood-made surface. The most significant peaks shown in the $y$-axis force plot are a consequence of the presence of obstacles in the robot trajectory.

### 7.6 Implementation and Experimentation

#### Concluding Remarks

In this chapter, hardware and software specifications of implemented VLs were presented along with experimental results and their analysis. VLs implementations were based on UML design to produce high quality software, which allowed its reuse in a modular way. Stability and synchronization of Petri Net designs were validated by analytical results obtained from experiments using implemented VLs and Internet connections with unpredictable and random delays.

The implemented VL for mobile robotics allows remote experiments on robotic navigation to be performed over the Internet. The path analysis showed the accuracy of the VL to make the robot follow remotely-planned paths. The time analysis showed unpredictable and large delays produced by the Internet connection. These delays did not affect stability and synchronization of the VL, since it was designed as an event-based system. Image transmissions over the Internet added large delays to the VL iteration cycles. VL performance was improved using a simple image reduction method (Appendix G). Furthermore, a complementary system to improve user observation capabilities was also implemented (Appendix E).

A VL for bilateral teleoperation was implemented using part of the previously implemented software. Java classes were reused as generic modules to implement the VL. This VL was designed and implemented as a real-time control system for Internet-based bilateral teleoperation of nonholonomic, differentially-driven robots. It provides the user with haptic and visual feedback to enhance the control experience by making it more natural and intuitive. VL behavior during a remote experiment showed that haptic feedback (force) is proportional to speed tracking errors produced by obstacle proximity and direction readings on a simulated workspace. Virtual forces around physical obstacles are generated by the system to provide sensory information concerning robot status in the remote environment. These experimental results validate the stability and synchronization of the closed-loop system to teleoperate the robot regardless of Internet delays.
Chapter 8
Conclusions

This research has generated a generic and modular model to develop Virtual Laboratories (VLs) for telerobotics over the Internet. The generic and modular model is translated into the standard Unified Modeling Language (UML) to create an object-oriented reference framework, which guides the development of VLs for telerobotics. The UML reference framework provides a skeleton, which is customized, based on experiment specifications. The customized framework dynamics is transformed into Petri Net notation. Petri Nets provide analytical methods to validate the stability of the designed VL over the Internet.

The development methodology takes into account necessary functionalities to mount, define, and execute a true experiment and prioritizes the sequential and synchronized execution of these functionalities in the VL design. Customized UML framework instantiations are generated to describe the structure and behavior of experiment functionalities. Each functionality behavior is captured by converting UML framework dynamics into the Petri Net modeling notation, which provides quantitative and qualitative analytical methods to validate appropriated behavioral properties in dynamic designs. These behavioral properties are related to design requirements for stable event-based telerobotic systems over the Internet. The validated Petri Net designs are merged into a composed net, which is analyzed to find undesired system states. These undesired states are then solved, using supervisory control for Discrete Event Systems (DES) to coordinate and synchronize functionalities and execute them sequentially to achieve true experimentation goals in the VL design.

The resulting UML framework and Petri Net design compose the complete structure and appropriate dynamics to model a VL application for telerobotics. These UML and Petri Net models guide the VL implementation in high level programming languages, based on the object-oriented paradigm. The structural design in UML constitutes the hierarchical and modular object-oriented representation of VL component instantiations, their inherent operation methods, interrelationships among instantiations, and component attributes. This UML design models procedures and devices to implement experiment functionalities into the VL application. The
composed Petri Net design provides a validated description of the system behavior to meet with the event-synchronization property for event-based telerobotic systems, and to sequentially perform experiment functionalities to provide the user with a true experimentation experience. Furthermore, this dynamic design is also compatible with event and message implementation in object-oriented programming languages.

The major result of this thesis is the validation of the proposed model and its development methodology as a suitable approach to systematically create the design of VL applications for telerobotics. This result was achieved by generating practical descriptions for developers, and validating the appropriate behavior to accomplish experiment functionalities and event-based design requirements for stable telerobotic systems over the Internet.

A VL application for mobile robotics was implemented using structural and dynamic designs generated by development methodology. Later, a new VL application for bilateral teleoperation was developed taking advantage of previously implemented modules. Experiments were carried out via the Internet and Internet2, using test beds that included sites at Monterrey N. L. and La Paz, B. C. S., México, geographically separated by 1,050 km.

8.1 Contributions

The contributions of this thesis can be outlined as follows:

- Creation of a generic and modular model for VLs for telerobotics. This model is converted into a reference framework described by a standard modeling language to lead the rational development of VLs from abstract designs to their implementation by means of the object-oriented paradigm.

- Formulation of systematic phases integrated into a development methodology. These phases generate the complete modeling of VL systems. Multiple diagrams, emphasizing different aspects of the design, are provided to describe the complexity of VLs in a comprehensive standard notation for developers and programmers.

- Formal formulation of a conversion function to translate UML statechart diagrams into Petri Nets. This conversion function allows translating statechart diagrams behaving as finite state machines into an equivalent ordinary Petri Net. The Petri Net inherits statechart properties and generates the same regular language.

- Definition and validation of design requirements in the dynamic modeling of VLs. Petri Net behavioral properties are related to event-based telerobotics design requirements for VLs. Event-based design requirements are validated, based on the coverability tree analysis of Petri Nets.
8.2. **Future Directions**

During the course of this work, some insights suggest possible future research directions. The future work relates to some topics researched as part of this project and also to some topics that were not investigated. The following are the main suggested future research directions:

- Additional research should be done relating to experiment management from a distance learning perspective. User performance could be evaluated by supervisors or intelligent tutors to provide a way to design new experiments according to the user's skills. This research work will enhance the behavioral design part in the proposed development methodology.

- The proposed model and its methodology provide a way to develop, design, analyze, and implement VLs. Future work can take this systematic approach as their base and generate software to automatically develop new VL applications.
Chapter 8. Conclusions

- VLs have much potential for the use of: multi-agent systems to provide Internet-based collaboration, task synchronization and error recovery; neural networks to adapt the event-based reference to a changing system or environment; genetic algorithms to generate optimal solutions and plan multiple tasks in telerobotic experiments.

- Quality of service (QoS) regarding real-time sensory feedback provided to the user by VLs should be studied. This research must take into account the development of measurement for sensory information, which should be provided by VL applications performing true experimentation, and the way QoS real-time sensory feedback can be guaranteed over the Internet using new communication protocols.

- Further research regarding VL systems with multiple Guests and Hosts should be done. This research should mainly extend the dynamic design of VLs to provide a systematic way to model VLs with multiple Guests, implying several users collaborating in one experiment, and multiple Hosts controlling various robotic devices. In this context, the event-based design theory for VLs, introduced in this research work, has potential to be taken as a reference. Notice that the event-based teleoperation control theory has been used as the foundation for a multi-operator, multi-robot collaborative control and for a hierarchical perceptive frame for human/machine cooperative control over the Internet [Elhajj et al. 2000a, Elhajj et al. 2002b]. Furthermore, the proposed event-based design theory for VLs and the Petri Net notation are compatible with real-time, event-priority features of middleware technologies, such as CORBA, to support multiple interactions in a VL application [Bottazzi et al. 2002, Amoretti et al. 2003].

The suggested topics will improve research and could generate new research niches in Internet-based telerobotics. This work provides a formal definition and systematic development of VL applications, which can be taken as a theoretical basis to develop technological advances.
Appendix A

UML Software Component Definition

This appendix defines software components of the VL model using the UML formalism. These components are described using UML class and collaboration diagrams, which are useful to provide a generic definition of necessary class components and their associations to develop Guest and Host software.

A.1 Guest Software

Guest software contains several classes representing processes to provide the user with capabilities to remotely observe, supervise, and interact with the experiments. These classes are defined by UML class and collaboration diagrams describing their intrinsic components and interactions. Four subclasses are composing the Guest software class: Input Processing, Sensory Conversion, Simulation, Settings/Tasks Database, and Communication.

A.1.1 Input Processing

When the user introduces analog inputs into Input Devices, these inputs are converted into discrete signals, which are interpreted by the system as commands, simulator commands, or database commands (for searching, saving, or retrieving settings and tasks). This input interpretation procedure is usually performed by case or if-then statements. Input Processing components and operation methods are depicted in Figure A.1.

Discrete inputs recognized as commands are sent to the Host system. Inputs recognized as simulator commands are sent to the Simulation instantiation. Database commands make that previously introduced commands, usually remaining in computer memory, be saved in the Settings/Task Database instantiation or be retrieved from it and introduced into the system to recreate performed experiments. The Input
Appendix A. UML Software Component Definition

Processing collaboration diagram is shown in Figure A.2. This collaboration diagram details Input

A.1.2 Simulation

The VL framework considers a Simulation class to define processes in the Guest system that provide immediate sensory feedback. Once the Input Processing instantiation introduces simulator commands into the Simulation instantiation, simulated data is generated and sent to the Sensory Conversion instantiation closing a control loop in the Guest system. Simulation class components are shown in Figure A.3.

Typically, Simulation procedures are implemented as a tool to foresee the effects of applied commands on the real experiment. The Simulation class is associated with
the *Input Processing* class, it receives commands from the *Input Processing* to carry out actions to set and perform simulated tasks. Specific data acquired from real experiments can be introduced into *Simulation* procedures to synchronize both the real and simulated experiments. Furthermore, *Simulation* class can send simulated data to the *Experiment Control* class in the *Host* system to perform a task that was successfully simulated. The *Simulation* collaboration diagram is shown in Figure A.4.

Figure A.4: *Simulation* collaboration diagram.

Figure A.4 describes *Simulation* class associations and messages sent to other classes of the VL framework. The *Simulation* class contains methods and techniques
representing experiment control, environment, equipment, sensor devices, and their interactions among them.

In simulated VLs, *Simulation* procedures retrieve simulated models from the *Model Library* class of the *Host* system. These models contain information to synthesize graphically robotic devices and their surrounding environment to perform simulated experiments.

### A.1.3 Sensory Conversion

The *Sensory Conversion* processes receive discrete data and converts it into appropriate signals for output devices to generate sensory feedbacks, e.g., visual, haptic, acoustic, among others. Figure A.5 depicts the *Sensory Conversion* class to describe its components. *Sensory Conversion* interrelationships are represented by the UML collaboration diagram shown in Figure A.6.

![Figure A.5: Sensory Conversion class diagram.](image)

![Figure A.6: Sensory Conversion collaboration diagram.](image)
In *Sensory Conversion* procedures, sensory data is converted into appropriate signals to activate electromechanical effectors and transducers. Therefore, a data conversion operation method is considered to generate linear feedback signals. Curve interpolation or linear regression among other techniques could be contained in the data conversion procedure.

On the other hand, compressed sensory data (usually visual) received from the Host is sent to restoration procedures to recover its properties and be converted into video signals appropriate for display devices. Decompression techniques for image compression standard formats are included in restoration procedures, such as JPEG decoders.

Furthermore, both sensed and simulated sensory data can be combined to be simultaneously displayed or played to enhance user perception when performing a remote experimentation.

### A.1.4 Settings/Tasks Database

In the *Guest* system, the *Input Processing* class saves and retrieves settings and tasks in a *Settings/Tasks Database* class to recreate performed experiments. In general, three basic operations are done by the *Settings/Task Database* class: searching, to locate settings or tasks in the database; reading, to retrieve information from the database; writing, to save data into the database. A parser procedure is considered for database command interpretation. Settings/tasks are typically retrieved from the computer memory and saved in the database. Saved data is grouped into two sets: settings and tasks (user commands). The *Settings/Tasks Database* class components and their associations are depicted in Figures A.7 and A.8.

![Figure A.7: Settings/Tasks Database class diagram.](image-url)
A.2 Host Software

Host software contains class components that represent processes to remotely perform an experiment. The following four subclasses are enclosed in the Host software class: Sensed Data Processing, Experiment Control, Model Libraries, Experiment Database, and Communication.

A.2.1 Sensed Data Processing

Sensor devices introduce sensory information into the Host computer via its input ports, which are managed by software drivers that handle computer hardware. These drivers acquire and convert sensory information into a discrete representation (sensed data). If a massive amount of sensed data (typically visual information) is sent to the Guest system, then a data reduction process is used to compress sensed data to reduce its size or eliminating irrelevant data, such as JPEG decoders. A data analysis process obtains the most significant information (specific data), which is issued to the Experiment Control class procedures to close a control loop in the Host system. Specific data is sent to Simulation procedures in the Guest system to synchronize the simulated experiment with the real one. Sensed Data Processing class components are shown in Figure A.9, their associations are defined by the UML collaboration diagram depicted in Figure A.10.

Figure A.9: Sensed Data Processing class diagram.
A.2. Host Software

Communication

Figure A.10: Sensed Data Processing collaboration diagram.

A.2.2 Experiment Control

The Experiment Control class represents processes to regulate experiment behavior according to user commands and settings sent by the Guest system. The Experiment Control class diagram is depicted in Figure A.11 and its collaboration diagram is shown in Figure A.12.

Figure A.11: Experiment Control class diagram.

Commands sent from the Guest system to Experiment Control procedures are interpreted as straight commands or configuration settings. Straight commands are verified to guarantee that system security limits are not exceeded, and then these commands are sent to translation procedures to generate appropriate commands for mechanism and control devices. Configuration settings are introduced to control algorithms, which regulate experiment behavior. These control algorithms can also receive specific data from Sensed Data Processing procedures to close a control loop in the Host system. Data and commands temporally remain in the computer memory until they are not significant for the Experiment Control procedures. These experimental parameters can
be permanently keep into a Experiment Database to make a further analysis.

![Experiment Control collaboration diagram](image)

**Figure A.12: Experiment Control collaboration diagram.**

### A.2.3 Model Libraries

In simulated VLs, Simulation procedures retrieve model libraries via Media, which are stored in Host. These libraries are implemented in modeling programming languages, such as VRML (Virtual Reality Model Language). These models can be displayed graphically by the user interface in the Guest system. They represent abstractly laboratory equipment and include its operation methods, interrelationships, and surroundings. Therefore, experiment development is delimited by the specified behavior in these models. These model libraries take care of operation control in simulated tasks and can provide simulated sensory information. The Model Library class diagram is shown in Figure A.13, Model Library associations are depicted in Figure A.14.

![Model Library class diagram](image)

**Figure A.13: Model Library class diagram.**
A.2.4 Experiment Database

Experiment Database procedures can save and retrieve commands, configuration settings, and specific data to recreate performed experiments. Three basic operation methods are performed by the Experiment Database class: searching, to locate experimental data; reading, to retrieve experimental data from the database; writing, to save data in the database. Experiment data is obtained from the Host computer memory. Figure A.15 shows Experiment Database class components. Component associations are described by the UML collaboration diagram depicted in Figure A.16.

A.3 Communication

The Communication class includes procedures for Guest and Host systems to manage and share information between them through Media. Communication management
between \textit{Guest} and \textit{Host} systems is considered as peer-to-peer or distributed. In peer-to-peer communication mode, \textit{Guest} behaves as client requiring tasks or information to \textit{Host}. In contrast, \textit{Host} behaves as a server, fulfilling \textit{Guest} requests. Peer-to-peer mode is implemented using Internet protocols, such as TCP, UDP, HTTP, among others. In distributed mode, an elaborated communication management between multiple peers is performed. It is implemented using standard interconnection technologies called middleware, e.g., CORBA technology, to create an object-oriented distribution of telerobotic resources. However, these technologies are based on protocols used by the peer to peer mode. Figure A.17 shows the most significant components that integrate the \textit{Communication} class.

\textit{Communication} procedures transporting data between \textit{Guest} and \textit{Host} systems are depicted in Figure A.18. Data sent by the \textit{Guest/Host} transmission procedure is transmitted as packets through \textit{Media} (Internet) and received by the \textit{Host/Guest} reception procedure. Communication management procedures distribute received data between the \textit{Communication} class and \textit{Guest/Host} class components. Commands and settings are usually sent from \textit{Guest} to \textit{Host}. In contrast, sensory data is typically sent from \textit{Host} to \textit{Guest}. 

**Appendix A. UML Software Component Definition**

![Diagram of Experiment Database collaboration](image)

**Figure A.16:** \textit{Experiment Database} collaboration diagram.

![Diagram of Communication class](image)

**Figure A.17:** \textit{Communication} class diagram.
Figure A.18: Communication collaboration diagram.
Appendix B

VL for Mobile Robotics: Robotics and Vision Techniques

This appendix describes in detail techniques and algorithms implemented in the VL for mobile robotics. The first section explains robotics methods and algorithms implemented in the C-Space Creation functionality. The following section describes navigation methods and algorithms, implemented in the Path Planning functionality to generate obstacle-free trajectories. The last section presents computer-vision algorithms and the control method implemented in the Path Following functionality to generate robot movements.

B.1 C-Space Creation

The use of convex elements makes simple and efficient the C-Space calculation. Convex hull calculation is applied over several sets of points to generate Configuration Obstacles (C-Obstacles). The C-Space calculation is based on the Graham’s Scan algorithm [Graham 1972], which is complete and sufficient for this application since the set of points (obstacle’s vertices) is small.

B.1.1 C-Space Definition

Let A be a robot (simple polygon) moving through a bidimensional workspace, which is defined by the set \( S = \{O_1, ...O_t\} \) of obstacles. Robot configuration in a workspace is specified by the number of robot degrees of freedom (DOF). Then, the robot configuration \( A(x, y, \phi) \) in the bidimensional workspace is defined by the vector \((x, y)\) to establish its translation movement and \( \phi \) for its rotation movement (as seen in Figure B.1).

The parametric space of the robot A is called its Configuration Space (C-Space) and is denoted by \( C(A) \). A point \( p \) in this C-Space corresponds to the robot configuration \( A(p) \) in the workspace. Obstacles and the robot configuration are projected
on the \textit{C-Space} to generate configuration space obstacles (\textit{C-Obstacles}) denoted by \textit{CO}. Mathematically, this projection equals the Minkowski subtraction operation [Lozano-Pérez 1983] between \textit{O} obstacles and the robot \textit{A} (\textit{CO} = \textit{O} \ominus \textit{A}). Then, the robot \textit{A} and obstacles \textit{CO} are assumed to be convex. \textit{CO} obstacles are defined as the set of points in the \textit{C-Space} that are the corresponding configuration of \textit{A} intercepting the negative projection of \textit{O} obstacles. Therefore, the \textit{C-Space} generated by the obstacle expansion defines all the robot configurations, where the robot collides with obstacles. These configurations are denoted by \textit{C-Space}_{\text{forbidden}}(\textit{A}, \textit{S}), and the free-collision configurations are denoted by \textit{C-Space}_{\text{free}}(\textit{A}, \textit{S}). Therefore, the \textit{C-Space} is the sum of the \textit{C-Space}_{\text{forbidden}}(\textit{A}, \textit{S}) and \textit{C-Space}_{\text{free}}(\textit{A}, \textit{S}).

A trajectory for the robot is projected as a curve on the \textit{C-Space} and vice versa; every robot configuration on the path is projected as a corresponding point on the \textit{C-Space}. A free-collision path is projected as a curve on the \textit{C-Space}_{\text{free}}. Since the mobile robot has a circular shape, it lacks of orientation. Therefore, the \textit{C-Space} is generated using one angular configuration.

\subsection*{B.1.2 \textit{C-space} Computation}

A domain \textit{D} on \(R^2\) is a convex set, if every two points \(q_1\) and \(q_2\) in \textit{D}, a segment \(q_1q_2\) is contained in \textit{D} [Preparata and Shamos 1985]. A point \(p\) of a convex set \textit{D} is an extreme point, if there is a line passing through \(p\) that all the points are aside of the line or over the line. Then, the convex hull is the minimal domain \textit{D} in \(R^2\) covering a set of \(n\) points. Given the set of points on the plane, Graham’s scan computes their convex hull.

Graham’s algorithm works in three phases:

1. Find an extreme point. This point will be the pivot, is guaranteed to be on the hull, and is chosen to be the point with the largest \(y\) coordinate.
2. Sort the set of points in order of increasing angle about the pivot. The algorithm
phase end up with a star-shaped polygon (one in which one special point, in this
case the pivot, can “see” the whole polygon).

3. Build the hull, by marching around the star-shaped polygon, adding edges when
we make a “left turn”, and back-tracking when we make a “right turn”.

Consecutive triads of points are analyzed to determine if they define a positive
or negative angle based on the determinant operator of its matrix coordinates in Eq.
B.1. If the internal angle of \( p_{i-1}, p_i, p_{i+1} \) is negative, then the three points make a
“right turn”, otherwise a “left turn”.

The three-point set with a middle point \( p_i = (x_i, y_i) \), where \( i = 2, n - 1 \), makes the
determinant:

\[
\det(p_{i-1}p_ip_{i+1}) = 2A = \begin{vmatrix} x_{i-1} & y_{i-1} & 1 \\ x_i & y_i & 1 \\ x_{i+1} & y_{i+1} & 1 \end{vmatrix} \tag{B.1}
\]

It represents the signed area \( A \) of the triangle defined by \( (p_{i-1}, p_i, p_{i+1}) \)
[Preparata and Shamos 1985]. The Graham algorithm is shown in Pseudo code
3.1:

**Pseudo code 3.1:**

**Procedure:** *Find_Convex_Hull*

1. Extreme_Point = *Find_Extreme_Point*(Points)
2. Sort_Points(Points, Extreme_Point)
3. For \( i = 1 \) to \( i < \text{Total_Points} \)
4. If \( \det(\text{Points}[i-1], \text{Points}[i], \text{Points}[i+1]) > 0 \) Then
5. \( \text{Add_Hull_Point}(	ext{Points}[i]) \)
6. \( i = i + 1; \text{Continue} \)
7. Else
8. \( \text{Remove_Point}(	ext{Points}[i]) \)
9. End If
10. End For

This algorithm uses the following elements:
Points: array of captured points.
Extreme_Point: reference point used to sort the point array.
Total_Points: total number of points.
*Find_Extreme_Point*(Points): finds a extreme point with largest \( y \) coordinate.
\( \det(p_{i-1}, p_i, p_{i+1}) \): calculates the determinant value of Eq. B.1.
*Add_Hull_Point*(Points[i]): adds the corresponding \( i \) point to the stack containing
points inside the convex hull.
*Remove_Point*(Points[i]): remove the corresponding \( i \) point from the stack.
When the obstacle expansion is complete, the C-Space is discretized in a rectangular grid (map from $\mathbb{R}^2$ to $\mathbb{Z}^2$). Where grid elements $(p_i=(x_i,y_i))$ are related to $\text{color}(p_i)$ values to tag every configuration found in the C-Space using the following evaluation function:

$$\text{color}(p_i) = \begin{cases} 1 \text{ (white), } & \text{if } p_i \in C-Space_{\text{free}} \\ 0 \text{ (black), } & \text{otherwise} \end{cases}$$ (B.2)

The subset of configurations associated with value 1 is the discretized $C-Space_{\text{free}}$. Configurations $q_{\text{start}}$ and $q_{\text{goal}}$ and all points in the trajectory should belong to the discrete $C-Space_{\text{free}}$.

### B.2 Path Planning

Path planning procedures define the shortest path between the starting point ($q_{\text{start}}$) and goal point ($q_{\text{goal}}$). The simple Navigation Function (NF1) is used, since it is efficient and simple to implement for a bidimensional workspace [Latombe 1991]. First, the discrete C-Space generated by the C-Space creation procedures is filled by the NF1 algorithm. The NF1 algorithm relates discrete $C-Space_{\text{free}}$ elements to a Manhattan distance value [Skiena 1990] using a wavefront expansion procedure. The Manhattan distance between two points, $p =(x_p,y_p)$ and $q =(x_q,y_q)$ is defined by:

$$d_4(p,q) = |x_q - x_p| + |y_q - y_p|$$ (B.3)

The wavefront procedure starts from $q_{\text{goal}}$ and then the path from $q_{\text{start}}$ to $q_{\text{goal}}$ is calculated by a gradient descent algorithm.

### B.2.1 Simple Navigation Function NF1

The NF1 algorithm works expanding a wavefront. The first element pushed into the data array is the $q_{\text{goal}}$ element. The algorithm takes out that element from the array and fills the corresponding position on a second array with a zero distance value, then it takes the element neighbors in vicinity-4 and assign their corresponding position with a distance value of one.

The algorithm continues taking out elements from the data array and assigns to the corresponding positions an incremental distance value. All neighbors that are not yet took out and are belonging to $C-Space_{\text{free}}$ are pushed into the data array end. In this way, the wavefront expands over the $C-Space_{\text{free}}$. Figure B.2 shows the numeric grid generated by the algorithm using a logarithmic scale.

The NF1 algorithm is described in Pseudo code 3.2. The algorithm ends when the $C-Space_{\text{free}}$ element subset accessible from the $q_{\text{goal}}$ element is fully explored. The algorithm is never trapped into local minimal values [Lengyel 1990].
Pseudo code 3.2:

**Procedure:** *Wavefront Expansion*(Goal)

1. *Add_Element_Not_Assigned*(Goal)
2. Distance[Goal] = 0
3. Dist = 0
4. **While** Element_List != *null*
5. Element = *Get_Next*(Element_List)
6. **For** each Neighbor of Element
7. **IF** Neighbor ∉ C-Space *forbidden* **Then**
8. Distance[Neighbor] = Dist + 1
9. *Add_Element_Not_Expanded*(Neighbor)
10. **End If**
11. **End For**
12. *Remove_Not_Expanded*(Element)
13. Dist = Dist + 1
14. **End While**

This algorithm uses the following elements:
- Element_List: list of matrix elements not assigned with a distance value.
- Distance[Element]: distance array obtained by the wavefront expansion.
- Neighbor: neighbor of the element been checked.
- *Add_Element_Not_Expanded*(Element): adds an element to the not expanded list.
- *Get_Next*(Element_List): gets the next element of list.
- *Remove_Not_Expanded*(Element): removes an element from the not expanded list.

Figure B.2: Numeric grid $E$ created by the NF1 planning procedure. The $E$ matrix describes the potential field used to plan the path. A logarithmic scale shows a cone shape on the destination point.
B.2.2 Gradient Descent Algorithm

Once the Manhattan distance values were assigned to each element of the \( C-Space_{\text{free}} \) in the numeric grid, a gradient descent algorithm finds an obstacle-free path between \( q_{\text{start}} \) and \( q_{\text{goal}} \). This path is obtained exploring the \( C-Space_{\text{free}} \) elements starting from \( q_{\text{goal}} \), and checking decrements in the assigned distance values until \( q_{\text{goal}} \) is attained. Each visited element is added to the point sequence that forms the path. When an element has several neighbors with the same distance value, the algorithm chooses any of them. This technique corresponds to a gradient descent of the distance value function generated in the \( C-Space_{\text{free}} \), see Figure B.3. The gradient descent algorithm is described in Pseudo code 3.3.

![Figure B.3: Path generated by the descent gradient algorithm on a 10^2 workspace grid, which was filled using the NF1 algorithm.](image)

Pseudo code 3.3:

**Procedure:** \( \text{Gradient Descent}(q_{\text{start}}) \)

1. \( \text{Element.Value} = \text{Get.Distance}(q_{\text{start}}) \)
2. \( \text{Add.To.Path}(q_{\text{start}}) \)
3. **While** \( \text{Element.Value} \neq 0 \)
4. **For** each Neighbor of Element
5. \( \text{Neighbor.Value} = \text{Get.Distance}(\text{Neighbor}) \)
6. **IF** \( \text{Neighbor.Value} < \text{Element.Value} \) **Then**
7. \( \text{Remove.Element}(\text{Element}) \)
8. \( \text{Element} = \text{Neighbor} \)
9. **End If**
10. **End For**
11. \( \text{Add.To.Path}(\text{Element}) \)
12. \( \text{Element.Value} = \text{Get.Distance}(\text{Element}) \)
13. **End While**
The elements used by the algorithm are the following:
Element.Value: distance assigned to the element.
Neighbor: neighbor of the current element.
Neighbor.Value: distance assigned to the neighbor.
Get.Distance(Element): gets distance assigned to element.
Add.To.Path(Element): adds an element to the path list.
Remove.Element(Element): removes an element from the not expanded list.

B.2.3 Curve Interpolation

The NF1 algorithm defines a path composed of continuous points, which follows obstacles' contour. Therefore, the planned path does not allow a smooth following by the robot [Scheuer and Xie 1999]. Then, the point sequence is analyzed to obtain its inflexion points, which are interpolated to a curve [Burden and Faires 1993] generating a smooth path.

To find these inflexion points consecutive point triads \((p_i, p_{i+1}, p_{i+2})\) on the continuous point sequence are analyzed to obtain inflexion points using a derivative-based method.

If \(m_2 - m_1 \neq 0\), where

\[
m_1 = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}
\]  \hspace{1cm} (B.4)

and

\[
m_2 = \frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}}
\]  \hspace{1cm} (B.5)

then \(p_{i+1}\) is added to the inflexion point list. These inflexion points are taken as control points to clamp a Spline curve generated by its interpolation.

The mathematical tool known as Natural Cubic Spline is used to interpolate inflexion points into a smooth curve without abrupt direction changes. This interpolation technique is based on cubic polynomials that conform a smooth curve under certain restrictions.

Curve approximation of an arbitrary set of points by Splines is based on the construction of different polynomials for each interval (point to point). In the same way, this approximation tries to accomplish certain characteristics as continuity and existence in its first and second derivatives on the interval, looking forward a smooth curve without intervals where the derivative do not exist (excepting in first and last interval). For the Spline curve calculation is necessary that the set of points behaves in an incremental way from an axis. Since the path is composed by a disorder of points, a vectorial approach is used. This approach uses the polynomial support as a calculation variable [Hernandez 1995]. The length of the polynomial support always
increases in a monotonic way. Using the accumulated distances between support points, we get an increasing sequence of values that can be used as an independent set of values to interpolate coordinates of the supporting points.

The Spline curve interpolation formulation is found in [Burden and Faires 1993]. To build the cubic Spline for a given function \( r = f(x, y) \), definition conditions are applied to the cubic polynomial with vectorial coefficients:

\[
S_j(r) = a_j + b_j(r - r_j) + c_j(r - r_j)^2 + d_j(r - r_j)^3
\]  
(B.6)

for each \( j = 0, 1, ..., n - 1 \).

Interpolation solution for the \( j \) -th point is the following linear equation:

\[
Ax = b,
\]  
(B.7)

where:

\[
A = \begin{bmatrix}
1 & 0 & 0 & ... & ... & 0 \\
h_0 & 2(h_0 + h_1) & h_1 & ... & ... & : \\
0 & h_1 & 2(h_0 + h_1) & h_2 & ... & : \\
: & ... & ... & ... & ... & 0 \\
: & : & : & h_2 & s(h_{n-2} + h_{n-1}) & h_{n-1} \\
0 & ... & ... & 0 & 0 & 1
\end{bmatrix},
\]  
(B.8)

\[
b = \begin{bmatrix}
0 \\
\frac{3}{h_1}(a_2 - a_1) - \frac{3}{h_0}(a_1 - a_0) \\
\vdots \\
\frac{3}{h_{n-1}}(a_n - a_{n-1}) - \frac{3}{h_{n-2}}(a_{n-1} - a_{n-2}) \\
0
\end{bmatrix},
\]  
(B.9)

and

\[
x = \begin{bmatrix}
c_0 \\
c_1 \\
\vdots \\
c_n
\end{bmatrix},
\]  
(B.11)
which has a unique solution for $c_0, c_1, ..., c_n$ and where $h_j$ is defined as:

$$ h_j = r_{j+1} - r_j. $$

Solving the linear equation $Ax=b$ gives coefficients $a_j$, $b_j$, $c_j$ and $d_j$ to substitute in the cubic polynomial that, in conjunction with the $r$ values from the starting point $(r_0, f(r_0))$ to the destination point $(r_{n-1}, f(r_{n-1}))$ constructs the interpolated curve.

Applied to the $x$-axis:

$$ S_{xj}(r) = a_{xj} + b_{xj}(r - r_j) + c_{xj}(r - r_j)^2 + d_{xj}(r - r_j)^3, \quad (B.13) $$

where:

$$ a_{xj} = x_j. \quad (B.14) $$

Applied to the $y$-axis:

$$ S_{yj}(r) = a_{yj} + b_{yj}(r - r_j) + c_{yj}(r - r_j)^2 + d_{yj}(r - r_j)^3, \quad (B.15) $$

where:

$$ a_{yj} = y_j. \quad (B.16) $$

Coefficients $b_j$, $c_j$, and $d_j$ are similar than the original Spline formulation in [Burden and Faires 1993]. This vectorial redefinition makes the cubic Spline independent of the sequence and space among clamp points, e.g. since $(r - r_j) = 0$ at any point $p_j$, and if $j = 2$ for $p_2$, as seen in Figure B.4,

$$ S_{2x}(r) = x_2, \quad (B.17) $$

$$ S_{2y}(r) = y_2, \quad (B.18) $$

written in vectorial notation

$$ S_2(r) = ix_2 + jx_2. \quad (B.19) $$

### B.3 Path Following

The path following procedure is performed once the path between $q_{\text{start}}$ and $q_{\text{goal}}$ is calculated. The path following procedure controls the robot using visual tracking. Workspace images are processed to find the robot landmark on the workspace to apply corrective actions to maintain the robot on the planned path. These corrective actions are applied as robot commands to generate discrete translational and rotational movements.

Images are grabbed in a gray scale to facilitate their processing. The image processing is composed of four procedures: image binarization, segmentation, characterization and matching. These procedures find and track an artificial landmark to know the robot position and its orientation. Then, a control procedure takes as a reference the planned path and compares the corresponding path point with the current robot position to generate robot discrete movements to follow the planned path.
Figure B.4: Path curve behaving in an increasing and decreasing way in both $x-y$ axes. The accumulative euclidean distance $r_j$ between path points $(p_j)$ allows performing the vectorial spline calculation, because is always an increasing parameter into polygonal functions. This Figure shows the corresponding vector for $p_2$.

### B.3.1 Image Definition and Binarization

A bidimensional image $I = f(x,y)$ is a matrix with $N$ elements (pixels) with discrete values of brightness or gray levels, as shown at the left side of Figure B.5. The gray level scale for $I \in [0, L-1]$ is $L=256$. $x$ and $y$ axes are natural numbers and their range is determinated by the image dimension: $x \in [0, n-1]$ and $y \in [0, m-1]$ for $I : (n \times m)$.

Binarization is an important process used to identify object’s extension allowing its shape analysis [Schalkoff 1989]. Binarization obtains a black and white image from a gray level image. This process uses a brightness threshold ($\mu_b$) to split image pixels into two classes: pixels taking a $L-1$ value (white) and pixels with a value of 0 (black),

\[
B(p_i) = \begin{cases} 
L - 1, & \text{if } f(p_i) \leq \mu_b \\
0, & \text{otherwise}
\end{cases}
\]  

where $p_i = (x_i, y_i)$.

The brightness threshold $\mu_b$ can be statistically defined based on the histogram analysis of pixels in the binarized image [Otsu 1979].

### B.3.2 Image Segmentation

The image segmentation procedure is the image partition in homogeneous and connected areas. This procedure separates objects from the image background. Each pixel is identified belonging to an object or to background.
The implemented image segmentation procedure is classified as a region-based technique [Leung and Malik 1998]. This technique produces partitions (called regions) based on several image characteristics, such as light intensity, color, and texture. The segmentation algorithm gathers similar and connected pixels, making a complete and systematic scan of every pixel of the binarized image. It uses an auxiliary matrix to keep a belonging register and relates each region found to a color.

The segmentation algorithm takes the binary image $B(x,y)$ as input, exploring it, pixel by pixel, from left to right and from top to down. Each pixel is assigned with a color associated to an object or to the image background. The algorithm works moving the three-pixel mask, shown in Figure B.7, through the complete image.

If mask element $p_c$ is found on a background pixel, then the algorithm continues scanning the pixel row. In contrast, if $p_c$ belongs to an object, then the color register is
updated. Several decisions are made by the algorithm to know if certain pixel belongs to an object or to the background. The region coloring algorithm is described in Pseudo code 3.4.

Figure B.7: Three-pixel mask used by the region coloring algorithm.

Pseudo code 3.4:

Procedure: Region_Coloring(B(x, y))

1. K = 1
2. For i = 0 to i < N
3.   If B(Pc) != 0
4.     If B( Ps) = 1 And B( Pi) = 0 Then
5.       ; descent color propagation
6.       Color( Pc) = Color( Ps)
7.     End If
8.     If B( Ps) = 0 And B( Pi) = 1 Then
9.       ; lateral color propagation
10.    Color(Pc) = Color(Pi)
11.    End If
12.    If B( Ps) = 1 And B( Pi) = 1 then
13.      If Color(Pi) = Color( Ps) Then
14.        ; indistinctly color propagation
15.        Color(Pc) = Color( Pi Or Ps )
16.    Else
17.      ; solve conflict between regions
18.      Combine_Regions()
19.    End If
20. End If
21. If B( Ps) = 0 And B( Pi) = 0 Then
22.    ; new color seed
23.    Color(Pc) = K
24.    K= K + 1
25. End If
26. Calculate_Descriptors( Pc)
27. End If
28. End For
The elements used by the algorithm are the following:

- **Color(Element):** assigns color to the binarized image element.
- **Combine_Regions():** if a conflict is detected between two regions, this procedure combines their information to convert them into a sole object.
- **Calculate_Descriptors():** Procedure executing the object characterization step, to calculate object descriptors.

### B.3.3 Object Characterization

Information concerning objects is gathered in an object array. This information contains quantitative data that helps to identify and characterize object's shape. This characterization computes object measures, such as perimeter, area, and some other mathematical relations. It is important that these descriptor characteristics must be invariant under scale, position and orientation changes. The following descriptor characteristics are used to identify the objects:

**Compacity.** Is a measure calculated based on object's area $A$ and its perimeter $P$ in a way to establish a measure of how slim or compact is the object shape, corresponding to the following relation:

$$C = \frac{P^2}{A} \quad (B.21)$$

**Hu first moments.** Several measures independent of scale, position and orientation variations were defined by [Hu 1962], and are known as Hu invariant moments. Given an image $f(x,y)$, Hu defined an algebra of invariant rotation moments.

In this work the first two moments were used:

$$\phi_1 = \eta_2 + \eta_{02}, \quad (B.22)$$

$$\phi_2 = (\eta_2 + \eta_{02})^2 + 4\eta_{11}^2, \quad (B.23)$$

where $\phi_1$ is known as the image quadrature, a relation measure between object's width and height, as shown in Figure B.8. While $\phi_2$ describes the shape diagonality.

The computation of invariant position and scale moments are required to calculate invariant rotation moments. Ordinary moments of order $p + q$ are defined by

$$m_{pq} = \sum_x \sum_y x^p y^q f(x, y) \quad (B.24)$$

Invariant translation moments known as centralized moments, are defined by

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q f(x, y), \quad (B.25)$$
Figure B.8: First Hu moment $\phi_1$, which represents object's quadrature. The object to be characterized and recognized is represented by the shadowed elliptical area.

where:

$$\bar{x} = \frac{m_{10}}{m_{00}}$$  \hspace{1cm} (B.26)

$$\bar{y} = \frac{m_{01}}{m_{00}}$$  \hspace{1cm} (B.27)

($\bar{x}, \bar{y}$) is the object centroid and $\mu_{00}$ its area. These descriptors are obtained in a linear incremental way as described in [Hu 1962]. Finally, normalized invariant scale moments are defined by

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^\gamma}$$  \hspace{1cm} (B.28)

where:

$$\gamma = \frac{p + q}{2} + 1$$  \hspace{1cm} (B.29)

The procedure that computes the aforementioned descriptor values is carried by the segmentation function: $\text{Calculate\_Descriptors}()$, in Pseudo code 3.4, line 21. The robot landmark is identified using its area and descriptor values. First, objects that have an area value under predefined threshold (noise) are eliminated. Then, descriptive values of the remaining objects are analyzed using an acceptance range ($\phi_{\text{max}} > \phi_{\text{min}}$). This range depends on the landmark descriptor variances obtained in prior experimentation.

### B.3.4 Robot Control

The path generated by the planning procedure is a sequence of clamp points of the interpolated curve. The path following task is done based on discrete robot movements to follow through the sequence, point by point, until $q_{\text{goal}}$ is reached. Discrete robot movements are generated by a control process that compares the current robot
position \( A(x, y, \phi) \) with the corresponding path point to reach \((P)\).

The characterization procedure identifies the robot landmark on the robot's top, which is composed by two circles: a big circle and small one. The big circle represents the robot's front, while the small indicates its rear. The robot position is indicated by the big circle centroid \((C_1)\) that is obtained using equations B.26 and B.27 from the object characterization procedure. Robot orientation \((\phi)\) is the angle defined by a line passing through both circle centroids \( C_1 = (\bar{x}, \bar{y}) \) and \( C_2 = (\bar{x}_2, \bar{y}_2) \), using the following trigonometrical function:

\[
\phi = \text{atan2}(\bar{y} - \bar{y}_2, \bar{x} - \bar{x}_2)
\]  

(B.30)

Then, angle \((\delta)\) is defined by a line passing through \( C_1 \) and the next path point \( P = (x_p, y_p) \) taking as reference the \( x\)-axis. It is calculated using the following function:

\[
\delta = \text{atan2}(\bar{y} - \bar{y}_p, \bar{x} - \bar{x}_p)
\]  

(B.31)

The control procedure computes the angular difference \((\theta)\) between \( \phi \) and \( \delta \) to define which movement (translation or rotation) is needed to minimize the distance between the robot and path point \( P \), e.g. an orientation command is sent to robot if a predefined threshold is greater than \( \theta \). A translation command is sent if the threshold is smaller than \( \theta \).

The translation and orientation commands are speed commands to set each robot wheel speed using a PID controller to control a PWM (Pulse Width Modulator) [K-Team 1998]. Orientation commands generate a discrete angular movement \( \Delta \phi \) to orientate the robot toward point \( P \). Translation commands generate a discrete forward movement \( \Delta d \) to decrease the distance between the robot and path point \( P \).

![Figure B.9: Trigonometrical method to generate angle \( \phi \) used by the control process to perform the path following task.](image_url)
The path follower algorithm is described in Pseudo code 3.5:

**Pseudo code 3.5:**

**Procedure:** Path_Follower()

1. \( A(x, y, \phi) = \text{Get-Robot-Position()} \)
2. \( \delta = \text{Calculate-Delta}((x, y), (x_p, y_p)) \)
3. \( \theta = \phi - \delta \)
4. If \( \theta < \) threshold Then
5. \hspace{1em} Forward(\Delta d)
6. Else
7. \hspace{1em} If \( \delta > \phi \) Then
8. \hspace{2em} If \( \theta > 180 \) Then
9. \hspace{3em} Turn(-\Delta \phi) ; left turn
10. \hspace{2em} Else
11. \hspace{3em} Turn(\Delta \phi) ; right turn
12. \hspace{1em} End If
13. \hspace{1em} End If
14. \hspace{1em} If \( \delta < \phi \) Then
15. \hspace{2em} If \( \theta > 180 \) Then
16. \hspace{3em} Turn(\Delta \phi) ; right turn
17. \hspace{2em} Else
18. \hspace{3em} Turn(-\Delta \phi) ; left turn
19. \hspace{2em} End If
20. \hspace{1em} End If
21. End If

The elements used by the algorithm are the following:

- **Get_Robot_Position():** obtains current robot position and orientation based on the image processing information.
- **Calculate_Delta((x, y), (x_p, y_p)):** calculates the \( \delta \) angle based on the trigonometrical equation B.31.
- **Forward(distance):** sends a forward command to the robot with the distance as argument.
- **Turn(angle):** sends an orientation command to the robot with the signed angular distance as argument (negative = left turn, positive = right turn).
Appendix C

VL for Bilateral Teleoperation: Robotics and Vision Techniques

This appendix describes in detail techniques and implemented algorithms for the VL for bilateral teleoperation, which were not described in Appendix B. The first section explains the robot position prediction procedure. The differential drive technique is explained in conjunction with the Khepera forward kinematics and its experimental characterization. The following section describes the potential field creation procedure. It explains the implemented algorithm to create a linear degraded potential field around obstacles. The last section presents the RLE (Run Length Encoding) algorithm used to compress transmitted images from Host to Guest.

C.1 Robot Position Prediction

Predicted position \( P_p \) is generated using the forward kinematics of the Khepera minirobot. The predicted position \( P_p \) is calculated taking into account the desired velocity \( V_d \), the current robot position \( P_k \), and an approximation of the distance traveled by the robot in a lapse of time. Since robot movements are controlled using the differential drive technique, its forward kinematics is simplified into several equations to predict robot movements.

C.1.1 Differential Drive

Differential drive is perhaps the simplest possible drive mechanism for mobile robots. As depicted in Figure C.1, a differential driven robot consist of two wheels mounted on a common axis controlled by separated motors.

Control of wheel velocities determine the robot’s motion. Under differential drive, each of the two wheels have to rotate about a point that lies on the common axis of the two drive wheels, to exhibit a rolling motion. Varying the velocity of the two wheels, the point of this rotation can be varied, and different trajectories can be generated.
Appendix C. VL for Bilateral Teleoperation: Robotics and Vision Techniques

Figure C.1: A differential drive controls robot pose by providing independent velocity control to left $v_l$ and right $v_r$ wheels.

At each instant in time, the point at which the robot rotates must have the property that the left and right wheels follow a path that moves the Instantaneous Center of Curvature (ICC) at the same angular rate $\omega$, and thus

$$\omega(R + l/2) = v_r$$

and

$$\omega(R - l/2) = v_l$$

where $l$ is the distance along the axle between the centers of the two wheels, the left wheel moves with the velocity $v_l$ along the ground and the right with velocity $v_r$, and $R$ is the signed distance from the ICC to the midpoint between the two wheels. Note that $v_l, v_r, \omega$ and $R$ are all functions of time. At any instance in time, solving for $R$ and $\omega$ results in

$$R = \frac{1}{2} \frac{(v_l + v_r)}{(v_r - v_l)}$$

$$\omega = \frac{v_r - v_l}{l}$$

A number of special cases are of interest. If $v_l = v_r$, then the radius $R$ is infinite and the robot moves in a straight line. If $v_l = -v_r$, then $R$ is zero and the robot rotates about a point midway between the two wheels (e.g., it rotates in place). This makes differential drive attractive for robots that must navigate in narrow environments. For other values of $v_l$ and $v_r$, the robot follows a curved trajectory about a point a distance $R$ away from the center of the robot, changing both the robot’s position and orientation.

A differential drive vehicle is very sensitive to the relative velocity of the two wheels. Small errors in the velocity provided to each wheel result in different trajectories, not just a slower or faster robot.
C.1.2 Forward Kinematics

Suppose that the robot is at some position \( P_k = (x_k, y_k, \theta_k) \), see Figure C.1. Through manipulation of \( v_l \) and \( v_r \), the robot be made to take in different poses. Determining the pose \( P_p = (x'_k, y'_k, \theta'_k) \) that is reachable given the control parameters is know as the forward kinematics problem for the robot. Since \( v_l, v_r, R, \) and \( \omega \) are functions of time, it is straightforward to show that if the robot has the position \( p_k \) at some time \( t \), and if the left and right wheels have ground-contact velocities \( v_l \) and \( v_r \) during the period \( t \rightarrow t + \delta t \), then the ICC is given by

\[
ICC = [x_k - R\sin(\theta_k), y_k + R\cos(\theta_k)]
\]  

and at time \( t + \delta t \) the pose of the robot is given by

\[
\begin{bmatrix} x'_k \\ y'_k \\ \theta'_k \end{bmatrix} = \begin{bmatrix}
\cos(\omega \delta t) & -\sin(\omega \delta t) & 0 \\
\sin(\omega \delta t) & \cos(\omega \delta t) & 0 \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix} x_k - ICC_x \\ y_k - ICC_y \\ \theta_k \end{bmatrix} + \begin{bmatrix} ICC_x \\ ICC_y \\ \omega \delta t \end{bmatrix}
\]  

Equation C.6 describes the motion of a robot rotating a distance \( R \) about its ICC with an angular velocity given by \( \omega \). By integrating Equation C.6 from some initial position \( p_k \), it is possible to compute where the robot will be at any time \( t \) based on the control parameters \( v_l(t) \) and \( v_r(t) \), that is, to solve the forward kinematics problem for the differential driven vehicle, as shown in Equations C.7, C.8 and C.9.

\[
x(t) = \frac{1}{2} \int_0^t [v_r(t) + v_l(t)]\cos(\theta_k(t))dt
\]  

\[
y(t) = \frac{1}{2} \int_0^t [v_r(t) + v_l(t)]\sin(\theta_k(t))dt
\]  

\[
\theta_k(t) = \frac{1}{2} \int_0^t [v_r(t) - v_l(t)]dt
\]  

Equations C.7, C.8 and C.9 describe a constraint on the velocity of the robot that can not be integrated into a positional constraint. This is known as a \textit{nonholonomic constraint} and is very difficult to solve in general, although solutions are straightforward for limited classes of the control functions \( v_l(t) \) and \( v_r(t) \). For example, if is assumed that \( v_l(t) = v_l, v_r(t) = v_r, \) and \( v_l \neq v_r \), then Equations C.7, C.8 and C.9 yields

\[
x_k(t) = \frac{l v_r + v_l}{2 v_r - v_l} \sin \left[ \frac{t}{l} (v_r - v_l) \right]
\]  

\[
y_k(t) = -\frac{l v_r + v_l}{2 v_r - v_l} \cos \left[ \frac{t}{l} (v_r - v_l) \right] + \frac{l v_r + v_l}{2 v_r - v_l}
\]  

\[
\theta_k(t) = \frac{t}{2} v_r - v_l
\]  

where \((x_k, y_k, \theta_k)_{t=0}=(0,0,0)\). If \( v_l = v_r = v \), then the robot’s motion simplifies to

\[
\begin{bmatrix} x'_k \\ y'_k \\ \theta'_k \end{bmatrix} = \begin{bmatrix} x_k + v \cos(\theta_k) \delta t \\ y_k + v \sin(\theta_k) \delta t \\ \theta_k \end{bmatrix},
\]  

(C.13)
thus the robot moves in a straight line, and if \(-v_l = v_r = v\), then Equation C.13 simplifies to
\[
\begin{pmatrix}
    x'_k \\
    y'_k \\
    \theta'_k
\end{pmatrix} = \begin{pmatrix}
    x_k \\
    y_k \\
    \theta_k + \frac{v}{l}
\end{pmatrix},
\]
(C.14)
thus the robot rotates in place.

### C.1.3 Khepera Characterization

The Khepera minirobot has a distance \(l = 53\) mm between the wheels, and two optical encoders that measure the instantaneous speed of each wheel. Encoder measurements are provided as pulses and each pulse equals a \(\frac{1}{12}\) mm step [K-Team 1998].

Introducing \(n_l\) and \(n_r\) for the number of pulses measured by the left and right encoders, the corresponding khepera distance value \(l\), and \(P_k=(x_k, y_k, \theta_k)\) as its initial pose with \(\theta_k\) in radians into the kinematic equations C.10, C.11, C.12, C.13 and C.14 yields

For \(n_l \neq n_r\)
\[
\begin{pmatrix}
    x'_k \\
    y'_k \\
    \theta'_k
\end{pmatrix} = \begin{pmatrix}
    x_k + \frac{53}{2} \frac{n_r + n_l}{n_r - n_l} \left[\sin\left(\frac{1}{636}(n_r - n_l) + \theta_k\right) - \sin(\theta_k)\right] \\
    y_k - \frac{53}{2} \frac{n_r + n_l}{n_r - n_l} \left[\cos\left(\frac{1}{636}(n_r - n_l) + \theta_k\right) - \cos(\theta_k)\right] \\
    \theta_k + \frac{1}{636} n_r - n_l
\end{pmatrix},
\]
(C.15)

For \(n_l = n_r\)
\[
\begin{pmatrix}
    x'_k \\
    y'_k \\
    \theta'_k
\end{pmatrix} = \begin{pmatrix}
    x_k + \frac{1}{12} n_r \cos(\theta_k) \\
    y_k + \frac{1}{12} n_r \sin(\theta_k) \\
    \theta_k
\end{pmatrix},
\]
(C.16)

For \(-n_l = n_r\)
\[
\begin{pmatrix}
    x'_k \\
    y'_k \\
    \theta'_k
\end{pmatrix} = \begin{pmatrix}
    x_k \\
    y_k \\
    \theta_k + \frac{|n_r|}{318}
\end{pmatrix},
\]
(C.17)

To generate an approximated predicted position \(P_k=(x'_k, y'_k, \theta'_k)\), it is necessary to characterize the Khepera robot. Table C.1 describes its average dynamic behavior obtained from several experimental tests with different speed settings traveling a straight line, starting from a static position.

From Table C.1 is deduced that the robot encoders will measure an average increment of 93.02 pulses/sec per 1 pulse/10 ms set on the PID-PWM controller of wheel motors. For instance, if the speed setting is \(d(13, 13)\), then each encoder will measure 1209.26 pulses per second. This experimental result indicates that the robot
C.2 Khepera Speed Controller Characterization

Since the VL for bilateral teleoperation was based on the event-based control approach, a detailed experimental characterization of the Khepera speed controller was required. As already mentioned in Chapter 6 and 7, the Khepera minirobot includes a PID controller to regulate a PWM (Pulse Width Modulator) used to control wheel motor speeds. This PMW sends pulses every 10 ms to both left and right wheel motors. This speed controller can be set using robot speed commands [K-Team 1998].

Several experimental tests were performed to characterize the robot speed controller. A Visual Basic application was implemented to control and measure the robot speed. These tests were done using 10 speed commands with settings ranging from 1 to 10 pulses/10 ms. The Khepera robot traveled a straight trajectory on the wood-made surface of the workspace for a lapse of time (500 ms). Figures C.2 and C.3 show measured speeds for both left and right motors acquired every 5 ms.

<table>
<thead>
<tr>
<th>Speed Setting</th>
<th>Avg. Time</th>
<th>Avg. Odom. Readings</th>
<th>Distance (cm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vl, Vr (pulse/10ms)</td>
<td>( \delta t ) (sec)</td>
<td>( V_{mleft}, V_{mright} ) (pulses)</td>
<td>(1 pulse = ( \frac{1}{12} ) mm)</td>
</tr>
<tr>
<td>d 1, 1</td>
<td>20.34</td>
<td>2002, 2004</td>
<td>16.68, 16.70</td>
</tr>
<tr>
<td>d 2, 2</td>
<td>14.54</td>
<td>2870, 2876</td>
<td>23.91, 23.96</td>
</tr>
<tr>
<td>d 3, 3</td>
<td>10.21</td>
<td>3023, 3031</td>
<td>25.19, 25.25</td>
</tr>
<tr>
<td>d 4, 4</td>
<td>6.20</td>
<td>2444, 2440</td>
<td>20.36, 20.33</td>
</tr>
<tr>
<td>d 5, 5</td>
<td>5.53</td>
<td>2729, 2725</td>
<td>22.74, 22.70</td>
</tr>
<tr>
<td>d 6, 6</td>
<td>5.26</td>
<td>3117, 3112</td>
<td>25.97, 25.93</td>
</tr>
<tr>
<td>d 7, 7</td>
<td>4.97</td>
<td>3425, 3428</td>
<td>28.54, 28.56</td>
</tr>
<tr>
<td>d 8, 8</td>
<td>4.33</td>
<td>3409, 3387</td>
<td>28.40, 28.22</td>
</tr>
<tr>
<td>d 9, 9</td>
<td>3.55</td>
<td>3139, 3136</td>
<td>26.15, 26.13</td>
</tr>
<tr>
<td>d 10, 10</td>
<td>3.30</td>
<td>3254, 3255</td>
<td>27.11, 27.12</td>
</tr>
<tr>
<td>d 11, 11</td>
<td>3.16</td>
<td>3448, 3447</td>
<td>28.73, 28.72</td>
</tr>
<tr>
<td>d 12, 12</td>
<td>2.61</td>
<td>3081, 3080</td>
<td>25.67, 25.66</td>
</tr>
<tr>
<td>d 13, 13</td>
<td>2.45</td>
<td>3132, 3129</td>
<td>26.10, 26.07</td>
</tr>
<tr>
<td>d 14, 14</td>
<td>2.51</td>
<td>3497, 3495</td>
<td>29.14, 29.12</td>
</tr>
<tr>
<td>d 15, 15</td>
<td>2.20</td>
<td>3288, 3285</td>
<td>27.40, 27.37</td>
</tr>
</tbody>
</table>

Table C.1: Dynamic behavior of the Khepera minirobot.

Using this approximation in conjunction with the robot forward kinematics equations, the desired speed \( V_d \) applied with certain \( \delta t \) and its current position \( P_k \), make it possible to predict the future robot position \( P_p \).

C.2 Khepera Speed Controller Characterization

will travel an average distance of 10.07 cm per second.
Figure C.2: Measured speeds for the right wheel motor.

From Figure C.2 and C.3, it can be deduced that the PID controller has an average tracking error of ± 1 pulse/10 ms for both wheel motors. In both speed plots, the PID controller is stabilized after a lapse of 90 ms. This lapse of time is used as a setting for the VL for bilateral teleoperation to generate a stable event-based bilateral interaction of the system with the user over the Internet. Notice that the event-based approach bases its design criterion on the previously mentioned Theorem 1 presented in Chapter 3.

C.3 Potential Field Creation

Once the C-Space is defined by the user, the displayed C-Space bitmap C is converted into a numeric matrix $M = m_{320,240}$. Matrix $M$ has a numeric potential field that simulates the C-Space according to the current obstacle configuration in the workspace and the robot shape. However, this matrix $M$ is introduced to another process to generate simulated proximity readings and create virtual forces around obstacles. These virtual forces are inversely proportional to the distance between the robot and the obstacle, and defines three navigation spaces, as shown in Figure 6.1.

The implemented algorithm used to build the potential field is described in Pseudo code C.1. The algorithm explores the matrix $M$ creating the decremented potential field around the obstacles depending on a previously established decrement step that can be calculated using linear, algorithmic or customized equations to simulate the infrared sensor response to obstacles with different colors and surfaces, see Khepera User Manual [K-Team 1998] for further information. Figure C.4 shows the numerical matrix $M$ created by the algorithm using a linear decremental step.
C.3. Potential Field Creation

Figure C.3: Measured speeds for the left wheel motor.

Pseudo code C.1:

Procedure: Create_Potential_Field

1. Create_Potential_Field(Potential1)
2. Potential2 = 0
3. For Potential1 to Potential1 ≥ 0
4. Potential2 = Potential1 - Step_Decrement
5. For c = 1 to c < Matrix_Width
6. For g = 1 to g < Matrix_Length
7. If (Cspc_Matrix[g + Matrix_Length × c]) ≥ Potential1 Then
8. If (Cspc_Matrix[(g + 1) + Matrix_Length × c]) < Potential2
9. Then Cspc_Matrix[(g + 1) + Matrix_Length × c] = Potential2
10. Else If (Cspc_Matrix[(g + 1) + Matrix_Length × (c - 1)]) < Potential2
11. Then Cspc_Matrix[(g + 1) + Matrix_Length × (c - 1)] = Potential2
12. Else If (Cspc_Matrix[(g - 1) + Matrix_Length × (c - 1)]) < Potential2
13. Then Cspc_Matrix[(g - 1) + Matrix_Length × (c - 1)] = Potential2
14. Else If (Cspc_Matrix[(g - 1) + Matrix_Length × c]) < Potential2
15. Then Cspc_Matrix[(g - 1) + Matrix_Length × c] = Potential2
16. Else If (Cspc_Matrix[g + Matrix_Length × (c + 1)]) < Potential2
17. Then Cspc_Matrix[g + Matrix_Length × (c + 1)] = Potential2
18. Else If (Cspc_Matrix[(g - 1) + Matrix_Length × (c + 1)]) < Potential2
19. Then Cspc_Matrix[(g - 1) + Matrix_Length × (c + 1)] = Potential2
20. Else If (Cspc_Matrix[g + Matrix_Length × (c + 1)]) < Potential2
21. Then Cspc_Matrix[g + Matrix_Length × (c + 1)] = Potential2
22. Else If (Cspc_Matrix[(g + 1) + Matrix_Length × (c + 1)]) < Potential2
23. Then Cspc_Matrix[(g + 1) + Matrix_Length × (c + 1)] = Potential2
24. End If
The elements used by the algorithm are the following:

- **Cspc_Matrix**: the C-Space numeric matrix $M$.
- **Potential1**: numerical reference used to establish the potential field maximum.
- **Potential2**: numerical reference used to establish the potential field.
- **Decrement_Step**: numerical reference used to decrement the potential field maximum.
- **Matrix_Length**: number of elements of the matrix length (320 elements).
- **Matrix_Width**: number of elements of the matrix width (240 elements).

![Three-dimensional image of the numeric potential field created by the algorithm.](image)

Figure C.4: Three-dimensional image of the numeric potential field created by the algorithm.

### C.4 Image Data Compression

When the workspace image is acquired by the *Host* system, it is converted into a discrete image $I=i_{320,240}$ using a 256-level gray scale. Then the image $I$ is compressed using a RLE-based algorithm before being transmitted to the *Guest* system. The RLE algorithm is described by Pseudo code C.2. Once the compressed image is received by the *Guest*, the inverse procedure is performed to recover the complete image, see Pseudo code C.3.

The implemented compression algorithm checks image elements to find repeated information. Since the image data is codified in 8 bits (256 grays), the algorithm
uses bit operations, such as \texttt{And} and \texttt{>>} (bit shift to right). The algorithm takes the image data from a byte array, then checks repeated data and uses two bytes to compress it. The first byte uses the most significant bit as a flag to indicate that contains the number of repetitions in the data array, and uses the following 7 bits to contain the number. The Second byte has a zero in its most significant bit and the following bits are the image data. Note that to obtain the maximum compression capacity (127 repetitions), the less significant bit of each image element is lost. However, the difference between two consecutive gray scales is not noticed by the user.

Pseudo code C.2:

\textbf{Procedure:} \textit{Compress\_Image}

1. \texttt{Compress\_Image(Image\_Array)}
2. \texttt{Index} = 0
3. \texttt{value} = 0
4. \texttt{Current\_Element} = 0x00
5. \texttt{Counter} = 0
6. \texttt{For} \texttt{c} = 0 to \texttt{c} < \texttt{Image\_Length}
7. \texttt{value} = \texttt{Current\_Element And} 0xff
8. \texttt{If ((Image\_Array[c] = Current\_Element) And (Counter < 127)) And (c < Image\_Length))}
9. \texttt{Then Counter} = \texttt{Counter} + 1
10. \texttt{Else}
11. \texttt{If (Counter = 0) Then do nothing}
12. \texttt{Else If (Counter = 1) Then}
13. \texttt{Compressed\_Image[index] = 0xff And (value >> 1)}
14. \texttt{Index} = \texttt{Index} + 1
15. \texttt{Else If (Counter = 2) Then}
16. \texttt{Compressed\_Image[Index] = 0xff And (value >> 1)}
17. \texttt{Compressed\_Image[Index + 1] = 0xff And (value >> 1)}
18. \texttt{Index} = \texttt{Index} + 2
19. \texttt{Else}
20. \texttt{Compressed\_Image[Index] = (Counter + 128) And 0xff}
21. \texttt{Compressed\_Image[Index + 1] = 0xff And (value >> 1)}
22. \texttt{Index} = \texttt{Index} + 2
23. \texttt{End If}
24. \texttt{Current\_Element = Image\_Array[c]}
25. \texttt{Counter} = 1
26. \texttt{End Else}
27. \texttt{End For}
28. \texttt{Compressed\_Length} = \texttt{Index}
29. \texttt{Return Compressed\_Image}
The elements used by the algorithm are the following:
Image_Array: byte array containing the image data.
Compressed_Imag e: byte array containing the compressed image.
Index: index for the compressed image array.
value: variable used to perform bit operations.
Current_Element: variable containing current repeated data.
Counter: number of repetitions.
Compressed_Length: length of the byte array containing the compressed data.
Image_Length: length of the original image array (76800 bytes).

Pseudo code C.3:
Procedure: Decompress_Imag e

1. Decompress_Imag e(Compressed_Imag e)
2. Index = 0
3. value = 0
4. Current_Element = 0x00
5. Counter = 0
6. For c = 0 to c < Compressed_Length
7.   If (Compressed_Imag e[c] And 0x80) ≠ 0
8.     Counter = Compressed_Imag e[c] And 0x7f
9.     value = Compressed_Imag e[c + 1] << 1
10.    c = c + 1
11.   For x = 0 to x < Counter
12.     Image_Array[Index] = value
13.     Index = Index + 1
14.   End For
15. Counter = 0
16. Else
17.   value = Compressed_Imag e[c] << 1
18.   Image_Array[Index] = value
19.   Index = Index + 1
20. End If
21. Return Image_Array

The elements used by the algorithm are the following:
Image_Array: byte array containing the decompressed image data.
Compressed_Imag e: byte array containing the compressed image.
Index: index for the decompressed image array.
value: variable used to store data bit operations.
Counter: number of repeated data in the compressed byte array.
Compressed_Length: length of the byte array containing the compressed data.
Appendix D

Functionality Modeling Phases

This appendix describes the functionality modeling phases of both VL developments for mobile robotics navigation and bilateral teleoperation. These modeling phases describe: the experiment functionality analysis, the experiment subsystem identification in the VL framework, and the conversion into a dynamic model and its analysis. The first section presents the modeling phases of the development of the VL for mobile robotics. The second section describes the modeling phases of the development of the VL for bilateral teleoperation. Both sections complete the VL developments described in Chapters 5 and 6. In this appendix, hierarchical charts and specification tables are generated and translated into customized UML frameworks. Framework dynamics is extracted (UML statechart diagrams) to be converted into Petri Nets. These Petri Nets are analyzed to validate their event-synchronization property as stable event-based designs over the Internet.

D.1 VL for Mobile Robotics Functionality Modeling

This section completes the development of the VL for mobile robotics navigation described in Chapter 5. The following subsections illustrate the modeling phases for the Path Planning and Path Following functionalities.

D.1.1 Path Planning Functionality Modeling

The Path Planning functionality is identified as the necessary procedures to define the robotic navigation experiment. This functionality defines specific independent variables to perform the robotic experiment. The Path Planning procedures define the obstacle-avoiding trajectory used as a reference for the Path Following procedures.

This subsection resumes the first three methodology phases, which are used to translate the Path Planning functionality into UML and Petri Net notations.
### D.1.2 Path Planning Functionality Analysis

The first phase of the development methodology provides a specification table and a hierarchical chart to describe procedure specifications and functions in the Path Planning functionality. Table D.1 extracts and summarizes the most significant Path Planning processes including functions and specifications.

Table D.1: The most significant specifications of the Path Planning functionality.

<table>
<thead>
<tr>
<th>Process</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>Displays GUI, image, C-Space and planned path</td>
</tr>
<tr>
<td>Mouse</td>
<td>Generates discrete signals for destination $(x_f, y_f)$ and acceptance</td>
</tr>
<tr>
<td>Visual Display</td>
<td>Processing of image $I$, C-Space bitmap $(P = 320 \times 240$ pixels), and path points ${(x_t, y_t), \ldots}$</td>
</tr>
<tr>
<td>Path Planning</td>
<td>Converts Bitmap $P$ into the C-Space matrix $C = {c_{320,240}}$, performs NF1 algorithm and gradient descent on $C$, and the Spline algorithm to generate a smooth trajectory ${(x_t, y_t), \ldots}$</td>
</tr>
<tr>
<td>User Interface</td>
<td>Gets destination coordinates $(x_f, y_f)$ and acceptation</td>
</tr>
<tr>
<td>Image Processing</td>
<td>Acquires NTSC image, generates matrix $I = {i_{320,240}}$, 256 grays, creates a binarized image $B$ from $I$, segments $B$ to create an object array $O = {o_1, o_2, \ldots o_j}$, performs characterization to identify robot’s artificial landmark $O = {o_1, o_2}$ and calculates robot’s position $(x_k, y_k, \theta_k)$</td>
</tr>
<tr>
<td>Camera</td>
<td>Captures workspace image and robot’s landmark</td>
</tr>
<tr>
<td>Robot</td>
<td>Khepera minirobot with artificial landmark.</td>
</tr>
<tr>
<td>Workspace</td>
<td>Workspace surface $(73.5 \times 98 \text{ cm}^2)$ with physical obstacles $(2 \times 3 \times 15 \text{ cm}^3)$</td>
</tr>
</tbody>
</table>

The next step in the experiment analysis phase generates a hierarchical chart from the specification table, as shown in Figure D.1. This chart groups the Path Planning functionality procedures in a hierarchical manner.

### D.1.3 Path Planning Subsystem Identification in the VL Framework

The second phase of the development methodology is the VL subsystem identification in the UML framework. Experiment functionalities are treated as subsystems and related to the VL framework in the first stage of the subsystem identification phase. This first stage is the structural modeling of VL subsystems to create UML class and
D.1. VL for Mobile Robotics Functionality Modeling

Figure D.1: Hierarchical chart of the Path Planning functionality.

deployment diagrams. This stage relates specification tables and hierarchical charts to UML class and deployment diagrams, taking as a reference the VL definition provided by the UML framework. Figure D.2 depicts identified VL classes of the Path Planning subsystem.

Figure D.2: Class diagram of the Path Planning subsystem.

The second stage of the subsystem identification phase is the VL subsystem dynamic modeling. This dynamic modeling procedure extracts relationships among identified VL classes, messages among them, and the execution sequence from the
process flow chart and the experiment description. The UML collaboration diagram with identified instances and messages is shown in Figure D.3.

![Collaboration diagram of the Path planning subsystem.](image)

Figure D.3: Collaboration diagram of the Path planning subsystem.

The second stage of the subsystem identification step also describes VL subsystem execution sequences. A detailed UML sequence diagram is described in Figure D.4.

The third stage of the subsystem identification phase generates UML statechart diagrams to present a detailed description of the Path Planning subsystem operation, based on the previously generated informal and formal definitions. This statechart diagram describe operations and internal states generated by the interaction of VL class instantiations, as shown in Figure D.5.
D.I. VL for Mobile Robotics Functionality Modeling

Figure D.4: Sequence diagram that describes the Path Planning subsystem.

Figure D.5: UML state diagram that describes internal states of the Path Planning subsystem.
D.1.4 UML-Petri Net Conversion and Analysis

The UML-Petri Net conversion phase transforms UML statechart diagrams into the Petri Net formalism to model and analyze the VL subsystem dynamics. Figure D.6 is the Petri Net representation of the Path Planning subsystem.

Figure D.6: Petri Net diagram that represents the Path Planning subsystem.
The generated Petri Net for the Path Planning subsystem, shown in Figure D.6, is a pure and ordinary Petri Net, which behaves as a marking graph and does not have conflicts in its structure. Therefore, this Petri Net exhibits a deterministic behavior. From the Petri Net coverability tree analysis depicted in Figure D.7 one can deduce that the VL subsystem is safe, all its transitions are firable, and that these transitions have one input and one output place (as a state machine). Furthermore, the coverability tree shows that transition \( t_{20} \) is \( L1 \)-live. Therefore, the VL subsystem reaches its goal state \((p_{21}, \text{path planned})\). This implies that the Path Planning Petri Net is event-synchronized.

\[
M = \begin{align*}
&\begin{array}{ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
Following functionality into UML and Petri Net modeling notations.

**D.1.6 Path Following Functionality Analysis**

The first phase of the development methodology provides a specification table and a hierarchical chart to describe specifications and functions of inherent procedures in the Path Following functionality. Table D.2 extracts and summarizes the most significant processes including functions and specifications.

<table>
<thead>
<tr>
<th>Process</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>Displays GUI, image, C-Space and planned path</td>
</tr>
<tr>
<td>Mouse</td>
<td>Generates discrete signal for start command</td>
</tr>
<tr>
<td>Visual Display</td>
<td>Processing of image $I$ to create C-Space bitmap ($P = 320 \times 240$ pixels), and path points ${(x_t, y_t), \ldots}$</td>
</tr>
<tr>
<td>User Interface</td>
<td>Gets start command</td>
</tr>
<tr>
<td>Image Processing</td>
<td>Acquires image NTSC, generates matrix $I = {i_{320,240}}$, 256 grays, creates a binarized image $B$ from $I$, segments $B$ to create an object array $O = {o_2}$, performs characterization to identify robot’s artificial landmark $O = {o_2}$ and calculates robot’s position $(x_k, y_k)$</td>
</tr>
<tr>
<td>Camera</td>
<td>Captures workspace image and robot’s landmark</td>
</tr>
<tr>
<td>Control Process</td>
<td>Applies control algorithms (comparison between $(x_k, y_k, \theta_k)$ and $(x_t, y_t)$ and threshold $\delta$), generates commands for robot $d(+v_l, +v_r)$, $d(\pm v_l, \mp v_r)$</td>
</tr>
<tr>
<td>Robot</td>
<td>Executes movement commands issued by the control process</td>
</tr>
<tr>
<td>Workspace</td>
<td>Workspace surface ($73.5 \times 98$ cm$^2$) with physical obstacles ($2 \times 3 \times 15$ cm$^3$)</td>
</tr>
</tbody>
</table>

The next step in the experiment analysis generates a hierarchical chart from the specification table, as shown in Figure D.8. This chart groups the Path Following functionality procedures in a hierarchical manner.

**D.1.7 Path Following Subsystem Identification in the VL Framework**

The second phase of the development methodology is the VL subsystem identification in the UML framework. Experiment functionalities are treated as subsystems and related to the VL framework in the first stage of subsystem identification phase.
This first stage is the structure modeling of VL subsystems to create UML class and deployment diagrams. The UML class diagram of the Path Following subsystem is shown in Figure D.9.
The second stage of the subsystem identification phase is the VL subsystem dynamic modeling. This dynamic modeling procedure extracts relationships among identified VL classes, messages sent, and the execution sequence from the process flow chart and the experiment description. The UML collaboration diagram with identified instances and messages is shown in Figure D.10.

Figure D.10: Collaboration diagram of the Path Following subsystem.

The second stage of the subsystem identification step also describes VL subsystem execution sequences. A detailed UML sequence diagram is described in Figure D.11.

Figure D.11: Sequence diagram that describes the Path Following subsystem.
The third stage of the subsystem identification phase generates UML statechart diagrams to give a detailed description of Path Following subsystem operations, based on the previously generated informal and formal definitions, as shown in Figure D.12.

![UML state diagram](image-url)

Figure D.12: UML state diagram that describes internal states of the Path Following subsystem.

### D.1.8 UML-Petri Net Conversion and Analysis

The UML-Petri Net conversion phase transforms UML statechart diagrams into the Petri Net formalism to model and analyze the VL subsystem dynamics. Figure D.13 is the Petri Net representation of the Path Following subsystem.

The Path Following Petri Net is a pure and ordinary Net. It has conflicts in its structure; therefore, the Petri Net exhibits a nondeterministic behavior. The conflict, with an OR-exclusive behavior, is presented in $p_3$ (control performed). From the coverability tree analysis of the Path Following subsystem, shown in Figure D.14, one can deduce that the system is safe and that all its transitions are firables. These transitions have one input and one output place as a state machine. Furthermore, the coverability tree shows that transition $t_{15}$ is L1-live, thus the subsystem reaches its goal state ($p_{15}, destination reached$). This implies that the Path Following subsystem is event-synchronized.
Figure D.13: Petri Net diagram of the Path Following subsystem.
Figure D.14: Coverability tree analysis of the Path Following subsystem.
D.2 VL for Bilateral Teleoperation Functionality Modeling

This section completes the development of the VL for bilateral teleoperation described in Chapter 6. The functionality modeling phases describe the Robot Teleoperation functionality using modeling notations (UML and Petri Net) and validate event-based control design requirements.

D.2.1 Robot Teleoperation Functionality Modeling

This subsection presents the first three phases of the methodology to translate the Robot Teleoperation functionality into a customized UML framework and a Petri Net. The first phase is the experiment design and analysis. In this phase, a specification table and a hierarchical chart are built. Table D.3 describes the most significant processes and specifications. Figure D.15 describes processes and functions in a hierarchical manner to describe these processes with their dependent functions.

D.2.2 Robot Teleoperation Subsystem Identification

The second phase of the development methodology has three stages to model the structure, dynamics, and operation of the Robot Teleoperation functionality. The first stage describes the structure using UML class diagrams, as shown in Figure D.16.
D.2. VL for Bilateral Teleoperation Functionality Modeling

Figure D.16: UML class diagram of the Robot Teleoperation subsystem.

The second stage of the identification phase provides UML collaboration and sequence diagrams. The UML Collaboration diagram with identified instantiations is shown in Figure D.17. A detailed UML sequence diagram is shown in Figure D.18.

The third stage models the detailed operation of the Robot Teleoperation subsystem and generates a UML statechart diagram, which is depicted in Figure D.19.

D.2.3 UML-Petri Net Conversion and Analysis

The third phase of the methodology converts the UML statechart diagram $ST$ into a Petri Net $PN$ using the conversion function $G(ST) = PN$. Figure D.20 represents the Robot Teleoperation subsystem. The Robot Teleoperation subsystem represented by the Petri Net shown in Figure D.20 is pure and ordinary. This Petri Net behaves as a non-marking graph and has a non-deterministic behavior. It presents a conflict in its structure; the subsystem can not control the number of iterations to reach the robot's destination. Just the user controls when robot movements will stop. This conflict, with an OR-exclusive behavior, is presented in place $p_9$. From the coverability tree analysis, shown in Figure D.21, one can deduce that the subsystem is safe and behaves as a state machine. The coverability tree shows transition $t_{24}$ as L1-live. Thus the Robot Teleoperation subsystem can reach its goal state ($p_{24}$, destination reached). This implies that the teleoperation subsystem is event-synchronized.
Figure D.17: UML collaboration diagram of the Robot Teleoperation subsystem.

Figure D.18: UML sequence diagram of the Robot Teleoperation subsystem.
D.2. VL for Bilateral Teleoperation Functionality Modeling

Table D.3: The most significant specifications of the Robot Teleoperation subsystem.

<table>
<thead>
<tr>
<th>Process</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor</td>
<td>Displays GUI, workspace image and predicted position $P_p$</td>
</tr>
<tr>
<td>Mouse</td>
<td>Generates discrete signal for start command</td>
</tr>
<tr>
<td>Joystick</td>
<td>Generates discrete signals that are translated as position $X_m$</td>
</tr>
<tr>
<td>Force Effector</td>
<td>Translates error vector $E$ into a force $F$</td>
</tr>
<tr>
<td>Force Conversion</td>
<td>Performs speed error calculation $V_d - V_m = E$</td>
</tr>
<tr>
<td>Visual Display</td>
<td>Image $I$ decompression and visual processing of $I$ and $P_p$</td>
</tr>
<tr>
<td>User Interface</td>
<td>Gets start command</td>
</tr>
<tr>
<td>Position Prediction</td>
<td>Predicts future robot position $P_p$ based on $V_d$ and $P_k$, using differential driven kinematics</td>
</tr>
<tr>
<td>Position Conversion</td>
<td>Converts the joystick signal into integer variables defined as the joystick position $X_m=(X_m, Y_m)$</td>
</tr>
<tr>
<td>Image Processing</td>
<td>Acquires and compresses image, generates matrix $I = {i_{320,240}}$, 256 grays, creates a binarized image $B$ from $I$, does segmentation on $B$ to create an object array $O = {o_j}$, performs characterization to identify robot’s artificial landmark $O = {o_2}$ and calculates robot’s position $P_k=(x_k, y_k, \theta_k)$</td>
</tr>
<tr>
<td>Communication</td>
<td>Transmits and receives signals between the User and Laboratory Computers</td>
</tr>
<tr>
<td>Speed Processing</td>
<td>Converts speed strings measures acquired from the robot’s odometer, into integer variables $(V_{mile}t, V_{mright})$</td>
</tr>
<tr>
<td>Camera</td>
<td>Captures workspace and robot’s landmark</td>
</tr>
<tr>
<td>Odometer</td>
<td>Captures the instantaneous speed from the robot wheels</td>
</tr>
<tr>
<td>Control Process</td>
<td>Generates and applies commands on the robot $d(±v_l, ±v_r)$, $d(±v_l, ±v_r)$</td>
</tr>
<tr>
<td>Robot</td>
<td>Executes movement commands issued by the control process</td>
</tr>
<tr>
<td>Workspace</td>
<td>Workspace surface ($73.5 \times 98$ cm$^2$) with physical obstacles ($2 \times 3 \times 15$ cm$^3$)</td>
</tr>
</tbody>
</table>
Figure D.19: UML statechart diagram of the Robot Teleoperation subsystem.
Figure D.20: Petri Net diagram of the Robot Teleoperation subsystem.
Figure D.21: Coverability tree analysis of the Robot Teleoperation subsystem.
Appendix E

Observer System

This complementary system controls a second mobile camera in an open-loop control mode. Figure E.1 describes the complete Observer system. The mobile camera is a Sony camera model EVI-D30, which is capable of making pan, tilt, and zoom movements. Images generated by the mobile camera are acquired by a video grabber board that compresses images in a JPEG (Joint Photo graphics Expert Group) format, using methods written in C and converted into JNI's (Java Native Interfaces).

Compressed images are sent from the Observer to the Guest computer via the Internet by means of a Java RMI (Remote Method Invocation) object server. Guest receives these images using a RMI client procedure. Received images are decompressed and displayed by a software application with a user interface, as shown in Figure E.2.

On the other hand, commands and Cartesian coordinates needed to control camera movements are introduced to another user interface. Commands and coordinates are sent to the Observer server via a socket-based TCP client procedure, and received by a listening socket-based TCP server procedure. Cartesian coordinates are transformed into polar coordinates. Both commands and positions are converted to the camera serial protocol called VISCA (Video System Control Architecture).

Performed tests indicated that there is a trade off between the JPEG image compression quality and the amount of data sent. The system got less than 8 Kbytes, when it compressed the image with a JPEG compression quality of 50.
Figure E.1: Complementary Observer system, it uses RMI and TCP communication procedures.

Figure E.2: User interfaces of the observer system. Left interface displays images and right interface allows the user to control camera movements.
Appendix F

VL for Mobile Robotics Improvements

Several changes to the initial design were made based on an experimentation stage. The final design layout of the experiment is shown in Figure F.1.

![Diagram of Virtual Laboratory for mobile robotics over the Internet](image)

Figure F.1: Final layout of the Virtual Laboratory for mobile robotics over the Internet.

The host autonomy and security were improved reducing image requests sent through the Media and using robot proximity sensors.
In the Guest system, the real-time user interaction with the experiment was also improved allowing insert virtual obstacles on the C-Space, avoiding “collisions” with the virtual obstacles, grabbing and dropping path modifications, allowing teleoperate the robot, and modifying experiment settings.

F.1 Using Robot Sensors to Avoid Physical Obstacles

Robot sensors were used to increase the system security. These sensors allow a dynamic C-Space, since these sensors can help to detect physical obstacles. They are helpful to avoid collisions with objects that were not described in the defined C-Space.

The robot has a command to obtain data from its sensors. Robot sensors measure the light reflected by obstacles, and the acquisition command consists of the “N” character followed by the carriage return symbol. The robot answer has the following format:

\[ n, \text{val}_\text{sens}.\text{left}.90^\circ, \text{val}_\text{sens}.\text{left}.45^\circ, \text{val}_\text{sens}.\text{left}.10^\circ, \text{val}_\text{sens}.\text{right}.10^\circ, \text{val}_\text{sens}.\text{right}.45^\circ, \text{val}_\text{sens}.\text{right}.90^\circ, \text{val}_\text{sens}.\text{back}.\text{right}, \text{val}_\text{sens}.\text{back}.\text{left} \]

This command reads 10 bit values of the eight proximity sensors, from the front sensor situated at the left of the robot, turning clockwise to the back-left sensor. Each value obtained from the robot answer has a range between 0 and 1023. Measures taken from back sensors are ignored due to the disturbances caused by the serial cable.

The sensed data processing is implemented based on [Diard and Lebeltel 1999] work. Sensory information obtained from the “N” command is processed to generate specific information. This sensory information gives direction and proximity of physical obstacles. Direction and proximity are helpful to avoid collide with physical obstacles, which were not considered in the C-Space.

Direction (Dir) and proximity (Prox) are calculated doing the sum of data given by the six front proximeters. Figure F.2 describes the behavior of Dir and Prox parameters. Dir calculation is done subtracting weighted measures from the left and right opposite sensors, therefore a positive or negative sign is obtained indicating the right or left side of the robot’s front, and a value to indicate the obstacle direction. The sum is divided by another sum of the data to normalize the results into a \(-10 < 0 < +10\) range.

\[
\text{Dir} = \left[ \frac{90(p_x5 - p_x0) + 45(p_x4 - p_x1) + 5(p_x3 - p_x2)}{9(1 + \sum_{i=0}^{5} p_x(i))} \right]
\]

\[(F.1)\]

Prox parameter is calculated based on the maximum value obtained from all prox-
F.2 Virtual Obstacles Insertion

In the original experiment the user had not possibilities to modify the real workspace once the C-Space definition was made. Building predefined virtual obstacles allows the user to modify the C-Space in real time. Virtual obstacles should be put on the C-Space when the path following task is performed. This characteristic gives as specification, the need of a fast building procedure to introduce virtual obstacles into the defined C-Space grid. Therefore, a new button in the user interface is displayed. If the button is pressed, then the next point indicated by the user on the workspace image will be a virtual obstacle. This point \((P_v = (x_v, y_v))\) is the centroid of a new “obstacle” with a predefined square shape, and size of 7 x 7 pixels. Then, the “obstacle” is expanded and drawn as C-Obstacle on the C-Space.

The C-Space grid is composed of 320 x 240 elements to represent C-Obstacles and the C-Space_free. Since the planning procedure needs a discrete grid to work on, then the numerical matrix \(E\) represents the C-Space grid.

The process to create the C-Space numerical matrix is highly time consuming. In the original experiment the matrix creation time is almost 9 seconds. Therefore, a
Appendix F. VL for Mobile Robotics Improvements

Figure F.3: Matrix $E'$ considers the shadowed area of the $C$-Space matrix $E'$.

A small matrix $(E')$ of $30 \times 30$ pixels is created. In this small matrix is calculated the virtual obstacle and its expansion. Then, the $E'$ matrix is put on the $E$ matrix, using the point $(P_v)$ as reference. Equation F.3 is used to introduce the $E'$ element values in the $E$ matrix.

$$e'_{ij} = e[x_v + y_v \times \text{Width}]$$  \hspace{1cm} (F.3)

Thus, the element that is in the left-up corner of the analyzed small area is

$$e'_{ij} = e[(x_v - 15) + (y_v - 15) \times \text{Width}],$$  \hspace{1cm} (F.4)

and the element in the right-down corner is described by

$$e'_{ij} = e[(x_v + 15) + (y_v + 15) \times \text{Width}].$$  \hspace{1cm} (F.5)

Using these equations, $e'_{ij}$ values of the $E'$ matrix are passed to the $E$ matrix. The time to perform this procedure is less than 10% of the time needed to create the $E$ matrix. Once introduced the virtual obstacle into $E$, the virtual obstacle is considered as another $C$-Obstacle and is taken in count to calculate new paths.

F.3 Avoiding Collisions with Virtual Obstacles

To avoid "collisions" with virtual obstacles, a new procedure to perform path verification was implemented. This verification procedure checks that planned path points $P_i$ are not in the $C$-Space$_{forbidden}$. This procedure verifies that path points $P_i$ are not falling in $e_{ij}$ elements with large values of the $E$ matrix. If a $P_i$ point is detected in the $C$-Space$_{forbidden}$, then a warning appears in the user interface to tell the user the need to do something to avoid it.
On the other hand, the system verifies that the robot has not passed the \( (P_i) \) point, which is inside of the \( C\text{-Space}_{\text{forbidden}} \). The current robot position \( (x_k, y_k, \theta_k) \) and the current index number \( (\text{index}) \) of visited path points are sent to the Guest system every time that a image is requested. If the index of the point \( P_i \) inside of the \( C\text{-Space}_{\text{forbidden}} \) is not already reached, then the Guest sends a command to the Host to stop two \( P_i \) points before the robot. If a possible "collision" is detected on the robot’s way, then the Guest system displays a warning in the user interface as is shown in Figure F.4.

![Figure F.4: Left image shows the path planned. Right image shows a virtual obstacle on the robot path. The warning message appears at the left-up corner of the image.](image)

**F.4 Modifying the Planned Path**

The user performs path modifications, when an obstacle is in the robot trajectory, by grabbing and dropping a point \( P_i \) with the PC mouse. The path array is changed two points before and two points after from the selected path point \( p_i \). A line defined by \( P_{i-2} \) and \( P_i \) is calculated and segmented. The same occurs with \( P_i \) and \( P_{i+2} \). Points obtained from both line segmentations excepting \( p_i \) are introduced into the cubic Spline interpolation procedure. If new points are needed to define the path modification, then these new points are inserted into the path point sequence. The path point sequence with its new segment is sent to Host, and the robot continues its path following task from the last reached path point, as is shown in Figure F.5.

**F.5 Robot Teleoperation**

A teleoperation interface was created to be accessible from the user interface, just clicking a button. This interface allows to the user teleoperate the robot in an open-loop control mode. The user sends commands to the Host system for forward, backward, turn right, or turn left the robot. These commands are executed depending
Figure F.5: Left image shows the robot stopped to avoid “collide” with a virtual obstacle on its path. In the right image, the user modifies the path to avoid the virtual obstacle, and the robot continues the path following task.

This delay is unpredictable, and the command action is almost always performed delayed.

On the other hand, commands are performed during a predefined lapse of time ($\Delta t$) and with a predefined speed value set on the robot. This lapse and speed value can be set using a button to configure the teleoperation mode. Time is set in milliseconds and speed is set as pulses per every 10 milliseconds. For instance, if time is set with a minimal value and a minimal amount of pulses is set to perform a forward command, then a minimal distance is traveled by the robot, and a vice versa effect is made with maximum settings.

Commands sent by the teleoperation control are straight commands for the robot. Therefore, if a teleoperation command is sent when the path following task is performed, the robot will perform the teleoperation command. Then, the robot will continue its following task trying to reach the next path point $P_{i+1}$ from its new position.

F.6 Adjustable Settings

Two buttons were also added to the user interface to let the user introduce teleoperation and automatic settings. Both buttons are shown in the user interface showed in Figure 7.3. The teleoperation configuration button presents an interface to introduce speed values and the lapse of time $\Delta t$ to perform forward, backward, and turn movements. Default settings are: 13 pulses/10 ms for forward and backward movements, and $\pm 4$ pulse/10 ms for turn movements.

The automatic configuration button allows the user to introduce a proximity threshold ($Prox$), the range of direction ($Dir$), the angular difference threshold ($\phi$), and the translational and rotational speed values. By default $Prox$ has 12, $Dir$ is in the
F.6. Adjustable Settings

Figure F.6: Time chart with two teleoperation commands sent to the Host system. Commands are received after a communication delay. The lapse of time and number of pulses for teleoperation commands can be set in the user interface. These settings determine how long is going to be the translational or rotational movement performed by the robot.

\(-5 < 0 < +5\) range, the forward movement has 13 pulse/10 ms and left and right turns have 4 pulse/10 ms.
Appendix G

Decreasing the Number of Transmitted Images

This appendix describes the method used to decrease the number of transmitted images from Host to Guest. The first section describes the method implemented in the VL for mobile robotics. It explains how the system performance was increased avoiding large time delays. The second section presents the dynamic designs (Petri Nets) of both VLs, using controller nets to decrease the number of transmitted images.

G.1 Implemented Method to Decrease the Number of Transmitted Images

To avoid delay because of continuous image requests by the Guest system, Guest starts performing 10 image requests and calculates their average delay ($T_{img}$). Then Guest divides the average delay by the Host execution time ($T_{host} = 131.145$ ms) to obtain the number of iterations of the Host system $n_{host}$ that can be executed during a lapse of time $T_{img}$.

$$n_{host} = \frac{T_{img}}{T_{host}} \tag{G.1}$$

When the Guest system starts its eleventh iteration sends a “p” character (2-byte length). Host reads this command and performs the path following task, without sending any response to Guest. Then, the robot performs its movement a specific lapse of time (60 ms). To travel a distance $\Delta d$ or rotate an angle $\Delta \phi$ and reach the next path point ($P_{i+n}$), the robot movements are held as a discrete event.

Guest sends $n - 1$ “p” characters and then an image request. Host sends the image data, and the “p” command cycle starts again.
Appendix G. Decreasing the Number of Transmitted Images

Figure G.1: Behavior of the VL for mobile robotics when the method to decrease the number of transmitted images is used. Starting from the eleventh cycle, the system overall loop is closed using 2-bytes length commands ("p"). These commands are quickly transmitted by Media instead of wait for an image every cycle. The path following task is performed depending on the number of "p" commands. Since there is no image delays, more distance is traveled by the robot in less time.

This method increases the VL performance when it executes the path following task. The robot moves $n_{\text{host}}$ times before been stopped because of the image delay. On the other side, the image displayed in the user interface is refreshed after $n_{\text{host}}$ cycles, and less data is passed through the Media.
G.2 Dynamic Models for both VLs

The design of the implemented method to decrease transmitted images essentially to the dynamic modeling. Therefore, just the Petri Net designs are described in this appendix for both the VL for mobile robotics and VL for bilateral teleoperation.

Figure G.2 shows the composed Petri Net design with a controller net that decreases the number of requested images by the Guest system. Transitions $t_{35}$, $t_{36}$, $t_{37}$ and $t_{38}$, and places $p_{34}$, $p_{35}$, and $p_{36}$ were added to the original composed Petri Net design for the VL for mobile robotics. $p_{36}$ is a place that can store $n_{host}$ tokens (signals) and has an arc with a weight of $n_{host}$ that fires $t_{33}$. Transition $t_{33}$ puts a token in $p_{37}$ that enables $t_{33}$ and disables $t_{33}$.

While $n_{host}$ tokens (system iterations) are not gathered, the Guest system sends "p" commands to Host and the user interface still displays the last transmitted image. When $n_{host}$ tokens are gathered, then the Guest system requests a new image from Host.

In the same manner, the VL for bilateral teleoperation was improved to reduce the number of images transmitted over the Internet. Although a RLE-based compression process was added, and a controller net was also added to reduce the number of transmitted images. Notice that haptic sensory information and simulated robot landmarks are provided in real time to the user.

Figure G.3 depicts the Petri Net containing an image reduction controller net to avoid large delays in the Host system. Transitions $t_{43}$, $t_{44}$, $t_{45}$ and $t_{46}$, and places $p_{40}$, $p_{41}$, $p_{42}$, $p_{43}$, $p_{44}$ and $p_{45}$ were added to the original composed Petri Net design for the VL for bilateral teleoperation.

In Host, place $p_{40}$ is a place to store $n$ tokens (iterations) and has an arc with a weight of $n$ to fire $t_{43}$. Transition $t_{43}$ puts a token in $p_{41}$ that enables $t_{33}$ and disables $t_{44}$. While $n$ tokens are not gathered, the Host system sends just the robot position and measured speeds to Guest. The user interface displays the last transmitted image, but displays simulated marks indicating the current and predicted robot positions. At the same time, the force feedback is played by the joystick. When $n$ tokens are gathered in $p_{40}$, then the Host system sends a compressed image to Guest, which displays the new image. At the same time, it provides the other sensory information to the user.
Figure G.2: Petri Net design used to decrease the number of transmitted images in the VL for mobile robotics.
Figure G.3: Petri Net design used to decrease the number of transmitted images in the VL for bilateral teleoperation.
Appendix G. Decreasing the Number of Transmitted Images
Bibliography


