A Formal Methodology for the Specification of Distributed Systems From an Object Perspective.

Sangbum Lee
Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/5392

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

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

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

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

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

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

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road Ann Arbor MI 48106-1346 USA
313 761-4700  800 521 0600
A formal methodology for the specification of distributed systems from an object perspective

Lee, Sangbum, Ph.D.
The Louisiana State University and Agricultural and Mechanical Col., 1992
A FORMAL METHODOLOGY FOR THE SPECIFICATION
OF DISTRIBUTED SYSTEMS FROM AN OBJECT PERSPECTIVE

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Sangbum Lee
B.Eng., Han Yang University of Korea, 1983
M.S., Louisiana State University, 1989
August 1992
Acknowledgements

First of all, I would like to express my sincere gratitude and appreciation to Dr. Doris L. Carver for the untiring advice, assistance and encouragement that she provided during this dissertation research. Her motivation, encouragement, and guidance led me through the years.

I also would like to thank the committee members: Professors David C. Blouin, S. Sitharama Iyengar, Donald Kraft and Bush Jones for serving and for the contributions they made to this research. In addition, I express my great appreciation to all faculty members and graduate students, particularly software engineering group students, in the department of the Computer Science. Association and discussion with them were an invaluable help to achieve my academic goal.

I wish to express my special thank to my parents and wife Mija Lee for their selfless support and heartfelt love. Without their generosity, love, patience and support, this work would have been impossible. I am always grateful to them.

Finally, I would like to give the best glory to God who always loves me.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>ii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td><strong>Chapter 1. Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Formal Methods</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Strength and Weakness of Formal Methods</td>
<td>4</td>
</tr>
<tr>
<td>1.2.2 Existing Requirements Specification Methodologies</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Scope of Research</td>
<td>9</td>
</tr>
<tr>
<td>1.4 Organization of Dissertation</td>
<td>11</td>
</tr>
<tr>
<td><strong>Chapter 2. A Formal Specification Language for Distributed Object-Based Systems</strong></td>
<td>12</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Related Works</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Related Formal Specification Languages</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2 Related Programming Languages</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Distributed Object-Oriented Systems</td>
<td>19</td>
</tr>
<tr>
<td>2.3.1 Distributed Systems</td>
<td>19</td>
</tr>
<tr>
<td>2.3.2 Distributed Object-Oriented Systems</td>
<td>20</td>
</tr>
<tr>
<td>2.3.2.1 Specification of Distributed Systems</td>
<td>22</td>
</tr>
<tr>
<td>2.4 A Specification Language, DOSL</td>
<td>23</td>
</tr>
<tr>
<td>2.4.1 Motivation behind Development</td>
<td>23</td>
</tr>
<tr>
<td>2.4.2 Major Features of DOSL</td>
<td>24</td>
</tr>
<tr>
<td>2.4.3 Informal Interpretation of a DOSL Specification</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>2.4.3.1 Definition Component</td>
<td>28</td>
</tr>
<tr>
<td>2.4.3.2 Body Component</td>
<td>28</td>
</tr>
<tr>
<td>2.5 Examples</td>
<td>30</td>
</tr>
<tr>
<td>2.5.1 Example 1: Producer-Consumer Problem</td>
<td>30</td>
</tr>
<tr>
<td>2.5.2 Example 2: Problem-Solving Organization</td>
<td>33</td>
</tr>
<tr>
<td>2.6 Syntax of DOSL</td>
<td>39</td>
</tr>
<tr>
<td>2.7 Summary</td>
<td>42</td>
</tr>
<tr>
<td>Chapter 3. Semantics of DOSL</td>
<td>44</td>
</tr>
<tr>
<td>3.1 Introduction to Semantics</td>
<td>44</td>
</tr>
<tr>
<td>3.2 Abstract Syntax</td>
<td>47</td>
</tr>
<tr>
<td>3.3 Operational Semantics of DOSL</td>
<td>49</td>
</tr>
<tr>
<td>3.3.1 The Transition System Method</td>
<td>52</td>
</tr>
<tr>
<td>3.3.2 The Petri nets Semantics</td>
<td>60</td>
</tr>
<tr>
<td>3.4 Examples</td>
<td>68</td>
</tr>
<tr>
<td>3.5 Supplementary Semantics of Message Passing Statements</td>
<td>74</td>
</tr>
<tr>
<td>3.5.1 The Synchronous Message Passing</td>
<td>75</td>
</tr>
<tr>
<td>3.5.2 The Asynchronous Message Passing</td>
<td>77</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>78</td>
</tr>
<tr>
<td>Chapter 4. Object-Oriented Analysis and Specification</td>
<td>80</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>80</td>
</tr>
<tr>
<td>4.1.1 Object-Oriented Development</td>
<td>80</td>
</tr>
<tr>
<td>4.1.2 Existing Methodologies for Object-Oriented Systems</td>
<td>82</td>
</tr>
<tr>
<td>4.2 Input Model</td>
<td>85</td>
</tr>
<tr>
<td>4.3 Building of the Knowledge Base Environment</td>
<td>91</td>
</tr>
<tr>
<td>4.3.1 Internal Representation of the Data Flow Diagram</td>
<td>91</td>
</tr>
<tr>
<td>4.3.2 Object Identification</td>
<td>92</td>
</tr>
<tr>
<td>4.3.3 Extracting Actions</td>
<td>94</td>
</tr>
<tr>
<td>4.4 Construction of an ER model</td>
<td>95</td>
</tr>
<tr>
<td>4.5 Regeneration of User Document</td>
<td>97</td>
</tr>
<tr>
<td>4.6 Construction of An Object Specification Model</td>
<td>97</td>
</tr>
<tr>
<td>4.6.1 Classification of Objects</td>
<td>98</td>
</tr>
<tr>
<td>4.6.2 Definition of the Object Model</td>
<td>99</td>
</tr>
<tr>
<td>4.6.3 Generation of the Object Model</td>
<td>102</td>
</tr>
<tr>
<td>4.6.4 Specification Model</td>
<td>104</td>
</tr>
</tbody>
</table>
Chapter 5. An Integrated Modeling Methodology for Distributed Systems .................................................. 109

5.1 Introduction ................................................................................................................................. 109
5.2 Input Models ................................................................................................................................. 110
5.3 Object-Driven Specification Methodology .................................................................................. 113
  5.3.1 Develop the Graphical Representations ................................................................................. 113
  5.3.2 Convert each Representation into Internal Form .................................................................... 117
  5.3.3 Build the Knowledge Base ...................................................................................................... 119
    5.3.3.1 Information from the Data Flow Diagram ......................................................................... 119
    5.3.3.2 Information from the State Transition Diagram ............................................................... 120
    5.3.3.3 Information from the Petri Nets ......................................................................................... 121
  5.3.4 Synthesis the Input ................................................................................................................ 121
  5.3.5 Generate an Object-Oriented Specification ......................................................................... 122
5.4 Summary ....................................................................................................................................... 126

Chapter 6. Summary .......................................................................................................................... 127

6.1 Summary of Results ..................................................................................................................... 127
6.2 Significance of Results .................................................................................................................. 129
6.3 Future Research ............................................................................................................................ 135

Bibliography ...................................................................................................................................... 137

Vita ...................................................................................................................................................... 147
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Related specification languages</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Primitive predicates for message passing</td>
<td>75</td>
</tr>
<tr>
<td>4.1 The level 01 data flow diagram's internal representation</td>
<td>92</td>
</tr>
<tr>
<td>4.2 The internal form of the data flow diagrams</td>
<td>93</td>
</tr>
<tr>
<td>4.3 Extracted objects from the level 02 data flow diagram</td>
<td>94</td>
</tr>
<tr>
<td>5.1 Requirements for an elevator system problem</td>
<td>114</td>
</tr>
<tr>
<td>5.2.1 Data flow diagram's internal format</td>
<td>117</td>
</tr>
<tr>
<td>5.2.2 Petri net's internal format</td>
<td>118</td>
</tr>
<tr>
<td>5.2.3 State transition diagram's internal format</td>
<td>118</td>
</tr>
<tr>
<td>5.2.4 The internal representation of a Petri net</td>
<td>118</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The waterfall model (software life-cycle)</td>
<td>3</td>
</tr>
<tr>
<td>2.1 The two message passing methods</td>
<td>27</td>
</tr>
<tr>
<td>2.2 Object module specification</td>
<td>29</td>
</tr>
<tr>
<td>3.1 The synchronous message passing</td>
<td>66</td>
</tr>
<tr>
<td>3.2 The asynchronous message passing without returning information</td>
<td>67</td>
</tr>
<tr>
<td>3.3 The asynchronous message passing with returning information</td>
<td>67</td>
</tr>
<tr>
<td>3.4 The Petri net representation of example</td>
<td>72</td>
</tr>
<tr>
<td>3.5 The Petri net representation of the producer-consumer problem</td>
<td>73</td>
</tr>
<tr>
<td>4.1 The overview of the object-oriented analysis technique</td>
<td>85</td>
</tr>
<tr>
<td>4.2.1 The high levels of data flow diagrams</td>
<td>89</td>
</tr>
<tr>
<td>4.2.2 The low level of data flow diagram</td>
<td>90</td>
</tr>
<tr>
<td>4.3 Actions</td>
<td>95</td>
</tr>
<tr>
<td>4.4 The entity-relationship diagram</td>
<td>96</td>
</tr>
<tr>
<td>4.5 Algorithm for the object identification</td>
<td>99</td>
</tr>
<tr>
<td>4.6 The general form of the object module</td>
<td>103</td>
</tr>
<tr>
<td>4.7 An algorithm for the object model construction</td>
<td>107</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.1.1 The data flow diagram of the elevator system</td>
<td>115</td>
</tr>
<tr>
<td>5.1.2 The state transition diagram for the elevator movement</td>
<td>116</td>
</tr>
<tr>
<td>5.1.3 The Petri nets for the elevator upward movement</td>
<td>116</td>
</tr>
<tr>
<td>5.2 Frame of an object module</td>
<td>120</td>
</tr>
<tr>
<td>5.3 The general format of an object module</td>
<td>123</td>
</tr>
<tr>
<td>5.4 An object module</td>
<td>124</td>
</tr>
<tr>
<td>6.1 The overview of the methodology</td>
<td>129</td>
</tr>
</tbody>
</table>
Abstract

Distributed computing systems are systems in which multiple processors run independently by communicating with each other. The design of distributed systems is difficult to achieve as the execution patterns of distributed system are typically more complex than those of non-distributed computing systems. The application of object-oriented techniques to the design of distributed systems has the potential to increase the power of modeling and computing. A formal methodology which includes a specification language, developed from an object perspective, for the development of distributed systems is presented.

The formal specification language, DOSL (Distributed Object-based Specification Language), represents the specification of distributed systems from an object perspective. DOSL has a hybrid format which combines the property-oriented approach and the model-oriented approach. In particular, it has strong features for message passing specification. The semantics of DOSL is defined formally by two operational semantics methods: transition systems and Petri nets.

In addition, a formal object-based methodology for the specification of distributed systems is given. The methodology presents a framework for using the DOSL specification language and includes an integrated formalized method for identification of objects, their operations and behaviors from multiple modeling formats. The implementation of the methodology is supported by assistance with a knowledge base.
Chapter 1

Introduction

1.1 Overview

As software systems become increasingly complex, many problems which have occurred in software system development have led to a situation known as a software crisis. The problems which occur during software development include an unexpected change of requirements, the large amount of information, and the length of the development process. Software engineering is a discipline which helps to address the software crisis by using various efficient methods and tools through all phases in software development [Pre87]. The primary goal of software engineering is to promote the development of software systems that are correct and efficient.

The process of software development can be explained as a model. One of the well-known conceptual models is called the waterfall model which represents the software life-cycle as five different phases, shown in Figure 1.1. In spite of criticism about its ill-match to the real world of software development, the waterfall model is widely used to explain graphically the process of software development. Among the five phases in the waterfall model, the requirements analysis and specification phase is debatably the most important because within this phase the requirements of the user must be specified precisely in order to guide the correct direction of software development. The document generated in this phase is called a specification which is used as a cornerstone for the remainder of the software development process.

Correct transferral of knowledge from the real world to the requirements specification is critical. In reality, a high percentage of software systems are discarded after implementation because they do not meet user requirements. Error-free requirements specification decreases the cost and effort of software development. Within the
requirements specification phase, the user's needs should be identified and analyzed correctly. The specifier starts the development of a proposed system by collecting information about requirements of the system from many resources. The requirements specification phase is typically initiated with a document, prepared by the user in a narrative format, in which the real world behavior of the system is described. This document becomes a primary source for the initial transformation of the real-world system information into the software development process. Based on this document, the properties and functionality of the system are frequently represented by semi-formal representative techniques, such as data flow diagrams. As the definition of the requirements proceeds, more formal methods, such as state transition diagrams and Petri nets, are used to show the dynamic control and behavior. However, as these various techniques represent different perspectives of the application, the system specifier must combine the requirements from representations of different modeling techniques to produce the specification.

There are inherent problems in the requirements specification phase which are primarily due to the complexity of the requirements and insufficient supporting methods. Many methods and tools, including CASE tools, have been developed or are currently under development for the requirements analysis and specification phase. The methods for this phase must include the power to efficiently derive the functionality and the constraints of the system. It is a current trend to combine methods through the entire software development process so that all phases can be covered by such integrated methods. Such a collection of methods is called a methodology [Oll83].

In this dissertation, a methodology is presented which helps the specifier to represent a specification from semi-formal graphical representations as a formal representation. It guides the specifier to identify objects, actions, relationships and behaviors from multiple graphical representations and integrates the information in such a way that an object model that encapsulates actions and specifies relationships among
objects is specified. A formal specification language which provides the formal mechanism for representing the initial object specification is defined. The methodology is designed for the development of distributed systems from an object perspective.

![Diagram](image)

**Figure 1.1 The waterfall model (software life-cycle)**

1.2 Formal Methods

The use of methods which can help eliminate tedious work improves the overall effectiveness of software development. A formal method, which has notations and rules based on mathematical concepts, is regarded as an approach which can improve
the software development process. Generally, formal methods are used in two areas: specification identification and program verification [Hal90]. As a system becomes larger and more complex, the need for formal methods increases, particularly in the requirements analysis and specification phase, because the generated specification in this phase directly affects the cost of software development, i.e., an inappropriate specification needs modification and feedback steps during software development, resulting in increased cost. A formal method typically has a language called a specification language which formally identifies the software system's behavior and structural properties. It is based on mathematical concepts, such as sets, relations, and functions and has its own syntax, semantics and mapping function just like other programming languages [Win90].

1.2.1 Strength and Weakness of Formal Methods

The use of a formal method provides numerous benefits for the development of a software system by eliminating ambiguity, incompleteness, and inconsistency from the initial user requirements and by representing behavior and properties of software systems formally and abstractly [Win90]. The advantages of using formal methods are discussed below.

First, ambiguity can be eliminated from user requirements when they are formally specified. The formal specification written with a formal specification language helps to denote the user requirements concisely and precisely so that incorrect trans­ferral of requirements can be reduced.

Second, a formal specification is used as a cornerstone for the software development process. It is possible to make a parallel development strategy. Testing is possible before the code is available. Generally, a formal specification makes it possible to perform formal verification, and test cases can be produced with a formal specification [Som89].
Third, improved maintenance of software systems is possible with a formal specification. The maintenance of the software is a major issue as the program size gets larger and more complex. Usually, software development is done by several separate groups. The designer and implementor can be different persons. To transfer the correct meaning of requirements, a formal specification, which can be interpreted unambiguously, is important.

Although formal methods provide many advantages in software development, they are not yet widely used. One reason is that the system designers are unwilling to employ the formal methods. The disadvantages of formal methods are described below.

First, the designers are often not accustomed to using formal methods. The novice system designer may be hesitant to use formal methods because specifying a formal specification with a formal specification language requires more time and effort than writing a program directly from an informal requirements specification.

Second, use of an inappropriate formal method makes it hard to define the system's behavior and properties precisely. Not all types of systems can be effectively applied with any arbitrary formal method. For example, if a specification language is used that does not provide for easy expression of the properties of the desired system, its use may ever make the specification more complex.

Third, while research has been done towards development of notations and techniques of the specification methods, tools for large scale specification are insufficient [Som89]. Formal methods which are suitable for small-size systems cannot be used for large-size systems directly. Therefore, methodologies and supporting tools which guide the specification for large size systems are needed.

In spite of these difficulties, use of formal methods for software system development is increasing and the development of formal specification languages is receiving
increased attention. In a recent paper, experiences of using formal methods for real projects are described in [Hal90]. The development of formal methods, particularly in requirements analysis and specification phase, is a timely and critical issue. An introduction to formal methods and methodologies is given in the following section.

1.2.2 Existing Requirements Specification Methodologies

Existing techniques for defining the requirements of software systems, particularly for functional requirements, are diverse. Conventional approaches to represent requirements analysis include formalized methods which specify both static and dynamic system behavior. The representative techniques for static behavior include regular expressions, algebraic axioms, and recurrence relations. Common methods to represent the dynamic description of systems include state-oriented techniques such as decision tables, transition diagrams, event tables and Petri nets.

As the field of requirements specification matures, many of the techniques described above are being used as a basis for the application of knowledge-based systems to the requirements specification phase. The global goal of automatic programming, which is the automatic generation of a system based on stated requirements, is a futuristic and ideal goal of software engineering. Much of the ongoing work towards this goal has its basis in artificial intelligence. Early work in automatic programming is found in [Bar79]. In [Bar79], the automatic implementation of abstract algorithms is discussed. In [Bar85a], application of domain specific knowledge to the automatic programming process in the Φ-nix project is described. In this project, the two difficult problems of transformations from informal to high-level formal specification and high-level formal to low-level formal were divided.

The divide and conquer approach has been followed by many researchers. Some researchers are working on the automatic generation of code from a specification [Gre82], [Fea82], [Bar85b] while other researchers are working on the formalization
of information extracted from an informal specification [Ric88], [Bal85], [Lub86], [Cor88b]. The need for informality in specifications is described in [Bal79] and the inherent problems with that informality is discussed in [Mey85].

Object-oriented approaches to specification have evolved. The Requirements Modeling Language, RML, is used to explicitly express requirements in a formal manner from a structured input. It does not have an automated translation to an implementation. RML models the real world from the object-oriented viewpoint, thus facilitating communication between the user and the analyst. An RML specification is written from the designer's perspective of the real world. The requirements model has its basis in semantic networks. RML is not suitable for the front end of the specification process, but use of a structured informal method such as SADT is recommended to precede the development of an RML model. The RML approach is not an automated process either for the development of the model or for the transformation of the model, but it represents a foundation for research directed toward the representation of requirements knowledge from a real world viewpoint [Gm82].

Gist is a formal specification language developed to emulate the natural language construction of specifications [Bal85]. It is a textual language based on object types, operations, and relationships of the object types. Gist has a syntax for expressing constraints on the behavior of the objects. A Gist specification is an operational specification that permits formal analysis. In order to enhance the readability of the Gist specification, a paraphraser recreates a natural language description of the specification to complement the validity checking process.

The knowledge-based system Kate [Fic88] provides assistance for the transformation of informal requirements to a formal specification. It is a domain-specific analysis technique utilizing the Glitter [Fic88] system to represent information needed to refine a specification. The refinements of informal requirements are criticized by using domain knowledge and usage scenarios.
The Knowledge-Based Requirements Assistant, KBRA, provides an incremental formalization of a system description [Czu88]. It is based on the idea that system development typically proceeds with support of a human assistant who pieces information together, notes inconsistencies, presents different viewpoints, and critiques the work. In KBRA, the specifier manually represents an abstract system description, data and domain specific information. KBRA is based on a cut and paste approach that operates form a library of reusable components. The user can interact with an intelligent notepad used to express requirements ideas. The notepad examines the natural language sentences, identifies verbs from a pre-defined list and identifiers items that are in the library. KBRA uses a presentation to display the evolving system description. The level of formality of the presentation ranges from informal to formal. The reasoning process is based on inheritance, automatic classification and constraint propagation.

Lubars [Lub86] and Tsai [Tsa88] present systems based on dataflow input representations. Lubars describes an intelligent environment, IDEA, for the specification and design phases of software development. It uses the dataflow diagram as its unifying representation model. It refines the dataflow specification based on refinement rules in the knowledge base. User input is required in the refinement process. The concept of reusability is addressed by an abstract graph representation of designs that can be selected to produce a specific design [Lub86].

Tsai [Tsa88] describes the Specification Transformation Expert System, STES. It provides intelligent assistance for the requirements phase by generating a structure chart design from a dataflow specification input. Heuristic rules expressing desirable design features such as coupling, cohesion, fan-in, and fan-out are contained in the knowledge base. These rules, coupled with designer interaction, are used to refine the specification into a design in a structure chart representation.
Each of these systems just described addresses only one aspect of the requirements specification process. The Requirement's Apprentice, RML Specification, Kate and KBRA have a common goal of developing a formal specification from various types of informal input. IDeA and STES represent knowledge-based systems that add formalism to the specification process but do not actually produce a formal specification. IDeA refines an informal DFD description and STES transforms a DFD description to a structure chart representation. As the systems become larger and more complex, the demand for knowledge-based CASE methods is increasing, an ideal methodology which fully supports automatic programming facilities still eludes researchers in software engineering.

1.3 Scope of Research

We have presented the need for formal methods in the realm of sequential computing. Such needs are also true in distributed computing systems in which multiple processors execute in parallel by message passing. In spite of inherent advantages, the development of distributed systems is difficult due to their complexity such communication and nondeterministic execution. One promising approach to increase the power of modeling in distributed systems is to adopt object-based techniques.

As the area of requirements specification evolves, increased research efforts are directed toward the application of knowledge-based systems to the requirements specification phase in order to expand the availability of automated assistance. Moreover, the requirement methodologies without formal specification languages lack the structure and formality necessary to allow mathematical analysis to support the goal of producing a complete, unambiguous, noncontradictory, requirements specification. Thus, a methodology which is centered around a formal specification language for distributed systems is the emphasis of this work.
This research began with a study of formal methods and formal specification languages. We investigated two promising areas, object-based approaches and distributed systems. Various methods and required features of object-based approaches were investigated. The study applied object-based techniques to the development of distributed systems. By providing an object-based perspective to the design of distributed systems, the modeling power is potentially increased. Benefits of capturing the requirements from an object-based perspective include the bounding of the primary system entities and the representation of the system in a manner that is amenable to communication with the user.

We limit the focus of this research to the specification of distributed systems. The objectives of this research are:

1) The development of a specification language which formally represents the specification of distributed systems.

2) The formal definition of the developed specification language.

3) The development of a methodology that assists with the derivation of a specification of distributed systems on the basis of object-based techniques.

Numerous modeling techniques for system development were investigated and several well-known modeling techniques among them are selected and used as front-end methods for this methodology. Representations of data flow diagrams, state transition digrams and Petri nets are used as input models because they are widely used for initial modeling of a desired system. Petri nets are used as a base model since it has properties which are suitable for describing the execution pattern of distributed systems. The knowledge base is included to provide some automated support to the methodology.
For the development of a specification language, existing specification languages and programming languages were analyzed in order to help determine the required features. The number of specification languages available for distributed systems is small; most existing specification languages do not support features such as message passing. Defining the formal semantics to a language is necessary to ensure that the language is sound. We provide an operational semantics for our language.

1.4 Organization of Dissertation

In this chapter, we have briefly introduced general problems in software development, formal methods, existing methods for the requirements specification, and our methodology for the specification of distributed systems. The outline of this dissertation is as follows. In Chapter 2, a formal specification language, called DOSL (a Distributed Object-based Specification Language), is introduced. The features of the language are discussed in detail with examples. In addition, DOSL is described with a formal syntax and semantics. The formal semantics of DOSL is given in Chapter 3 where the operational semantics of DOSL is presented.

The methodology is introduced in Chapters 4 and 5. In particular, Chapter 4 describes the technique used as a front method for the methodology. It derives a frame object-based specification model using two different levels of data flow diagrams. The expanded methodology, which builds on the technique described in Chapter 4, is discussed in Chapter 5. It derives a formal and precise specification from multiple modeling techniques for distributed systems based on an object-based perspective. The specification is derived by integrating multiple viewpoints of modeling techniques such as data flow diagrams, state transition diagrams, and Petri nets. The derived specification is specified in a generic format which provides a starting point for using DOSL to complete the specification. Finally, a summary and future research are presented in Chapter 6.
Chapter 2

A Formal Specification Language for Distributed Object-Based Systems

2.1 Introduction

A formal specification language is a high level, abstract language which may or may not be executable. It contains features which can identify the behavior, structural properties or/and constraints of software systems formally and abstractly. We use Wing’s definitions in [Win90]:

Definition 1: A formal specification language consists of three components, \(<\text{Syn}, \text{Sem}, \text{Sat}>\), where \(\text{Syn}\) and \(\text{Sem}\) are sets and \(\text{Sat} \subseteq \text{Syn} \times \text{Sem}\) is a relation between them. \(\text{Syn}\) represents the syntactic domain, \(\text{Sem}\) denotes the language’s semantic domain and \(\text{Sat}\) means its relation.

Definition 2: Given a specification language, \(<\text{Syn}, \text{Sem}, \text{Sat}>\), if \(\text{Sat}(\text{syn}, \text{sem})\), then \(\text{syn}\) is a specification of \(\text{sem}\), and \(\text{sem}\) is a specificand of \(\text{syn}\).

Specification languages differ from each other in their \(\text{Syn}, \text{Sem},\) and/or \(\text{Sat}\). They can be categorized in several ways depending on their formal methods. In general, there are two major types of formal methods: model-oriented and property-oriented [Win90]. They also can be classified into procedural abstraction and data abstraction [Geh85] or operational and definitional [Cor88a] according to the different viewpoints. In this dissertation, we will use Wing’s classification. In the property-oriented method, a specification is written indirectly with a set of properties, for example, in the form of a set of axioms. In addition, a specification of the property-oriented method is defined as an abstract data type which is characterized by the behavior of the operations [Geh85]. There are two subgroups in the property-oriented method: axiomatic approach and algebraic approach.

In the axiomatic specification technique, a specification is specified by a set of functions that is represented by pre-conditions and post-conditions to specify each
operation of the type [Win90]. Since this approach specifies the behavior of an abstract data type in terms of axioms, it is useful for object-oriented system development. In addition, this technique facilitates proving of programs using data abstraction [Geh85]. There exist various approaches in this technique. Generally a specification in the axiomatic technique consists of two sub-parts: one part specifies the interface and the other part specifies the system behavior. Iota and OBJ [Geh85], [Fut85] are examples of specification languages based on the axiomatic approach.

The algebraic specification technique, introduced by Guttag [Gut77], is widely used for specifying a system. The algebraic approach is also appropriate for object-oriented development, particularly for sequential systems, since the actions on an object are specified in terms of their relationships [Som89]. A specification in this approach emphasizes the constraints/conditions of a system. This approach is more understandable than the axiomatic approach because the behavior of operations is specified in a more program-like format. In general, an algebraic specification consists of three explicit parts: syntactic, semantic and restriction. The syntactic aspects are specified in the syntactic part; axioms which represent the behavior of each operation are defined in the semantic part; the limitations of axioms and pre-conditions are specified in the restriction part [Geh85]. A primary difficulty is that it is often not easy to derive axioms for the general application. In addition, this approach is difficult to apply to parallel/distributed systems because it is hard to specify communication aspects. In spite of these problems, the number of formal specification languages developed based on this technique are more prevalent than those of other techniques. One reason is that the axioms explain not only the definition but also the semantics and constraints. Specification languages such as Act One [Ehr85] and Larch [Win87] are based on the algebraic specification technique.

In the model-oriented method, a specification is described directly with mathematical notations. The change of state in a system is denoted explicitly using
mathematical operations. Generally, a model-based specification is more concise than other approaches because the detailed behavior inside a module is explicitly specified. Representative languages in this category are VDM [Jon80] and Z [Spi88] for sequential systems, and CSP [Hoa85] and Unity for concurrent/distributed systems.

In this chapter, we present a formal specification language, called DOSL (Distributed Object-based Specification Language), which is defined to specify the requirements of distributed systems based on an object-oriented perspective. DOSL has a hybrid format based on both the algebraic approach and model-oriented approach. While the environment information of an object is specified with the algebraic approach, the internal behavior of the object is denoted with the model-oriented approach. The primary goal of DOSL is to provide a formal specification language that specifies an object model by emphasizing communication and nondeterministic features of distributed systems. The outline of this chapter is as follows. Related formal specification languages are presented in Section 2.2. Distributed systems including distributed object-oriented systems are introduced in Section 2.3. Motivation and major features of DOSL are given in Section 2.4. In Section 2.5, example DOSL specification models are illustrated. The DOSL syntax is given in Sections 2.6. Section 2.7 contains the summary. The semantics of DOSL is described in Chapter 3.

2.2 Related Works

2.2.1 Related Formal Specification Languages

Specification languages for distributed systems have different semantic domains, i.e., they specify state sequences, event sequences, or state and transition sequences of a system [Win90]. Since the communication between modules is achieved by message passing, specifying communication aspects, including synchronization and parallelism between modules, is extremely important. Specification languages which have
been developed to specify the behavior of distributed computing systems include CSP, CCS, and Unity in the model-oriented approach. LOTOS and Lamport's transition axiom [Lam89] are examples of the property-oriented approach.

Several representative specification languages are summarized in Table 2.1, including DOSL. They are analyzed according to desirable features of formal specification languages including characteristics required for specifications, such as formalism and data abstraction. Many languages lack supporting features for distributed systems, such as message passing. CSP and LOTOS were developed for distributed/concurrent systems. Larch and OBJ are used for the development of sequential object-oriented systems. Two representative specification languages for distributed systems, CSP and LOTOS, are described below.

**CSP** :: Hoare [Hoa85] introduced a conceptual language called Communicating Sequential Processes (CSP) which combines guarded commands, parallel commands, and other primitive commands. He claimed that input, output and concurrency should be regarded as primitives of programming [Hoa85]. Many concurrent and distributed programming languages are designed based on CSP's concepts. Occam and Ada [Dod80] are representative CSP-based languages. CSP is regarded as one of the most attractive languages for modeling concurrent systems, but it has some limitations to use for object-oriented systems; that is, it does not explicitly support an abstract data type.

**LOTOS** :: LOTOS (Language for Temporal Ordering Specification) was developed to formally specify distributed systems under the ESPRIT/SEDOS project. LOTOS is an extensional description technique based on the temporal ordering of events using the concept of abstract data typing. The concepts are strongly affected by the algebraic specification language Act One, CCS and other languages.
Table 2.1 Related specification languages

<table>
<thead>
<tr>
<th>Application domain</th>
<th>LARCH</th>
<th>OBJ</th>
<th>CSP</th>
<th>LOTOS</th>
<th>DOSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base theory</td>
<td>Axiomatic &amp; Algebraic</td>
<td>Axiomatic</td>
<td>Model-oriented</td>
<td>Algebraic &amp; logic (FOL)</td>
<td>Algebraic &amp; Model-oriented</td>
</tr>
<tr>
<td>Executable</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Seq./Con.</td>
<td>Sequential</td>
<td>Sequential</td>
<td>Concurrent</td>
<td>Distributed</td>
<td>Distributed</td>
</tr>
<tr>
<td>Data abstraction</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Formalism</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Validation</td>
<td>Yes (semantic checking)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Object-oriented Support</td>
<td>Not fully</td>
<td>Not fully</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Related Lang.</td>
<td>Pascal, CLU</td>
<td>OBI-2, NPL, SPECINT</td>
<td>Occam.</td>
<td>CCS, Act One</td>
<td>ABCL/1, CSP</td>
</tr>
</tbody>
</table>

2.2.2 Related Programming Languages

The design of DOSL was influenced by several existing languages and logics. DOSL adopted some basic concepts and syntactic formats from a variety of approaches including temporal logic [Pnu77], guarded commands in CSP and ABCL/1 [Yon87], [Yon90]. A brief introduction to these concepts is given as follows.

ABCL/1:: ABCL/1 (An object-Based Concurrent Language 1) and ABCM/1 were developed to enable the construction of different fields of software systems based on
object-oriented computing and parallelism for the Japanese fifth-generation systems project [Yon90]. ABCM/1 is a derivative of an actor model and ABCL/1 is its descriptive language. In ABCM/1, each module has its own processing power and may have local persistent memory, the contents of which can only be accessed by itself. There are many similarities between the ABCM/1 model and the actor model [Agh86]. ABCL/1 is a pure distributed object-oriented programming language which has strong features for expressing communication patterns. There are three different message passing types: past, now and future. The past and future types are similar to asynchronous message passing and the now type is close to synchronous message passing. Moreover, ABCL/1 supports priority in messages, i.e., the express model message has a higher priority than an ordinary mode message. ABCL/1 adopts many syntactical features from LISP, even though it does not use LISP as a base language; thus, the overall format is very similar to LISP. DOSL has the syntactic format of the message passing statement similar to ABCL/1.

**Guarded Commands:** The guarded commands, presented by Dijkstra [Dij75], define alternative choices in nondeterministic ways. There are two statements; an alternative statement and a repetitive statement. The syntax of the alternative statement is:

```plaintext
if condition-1 \rightarrow statement-1, ...
\]
condition-2 \rightarrow statement-2, ...
\]
\]
condition-n \rightarrow statement-n, ...
fi
```

Each condition, called a guard, results in a true or a false value. If more than one guard is true, only one of them is chosen non-deterministically and the statement
on the right hand of "\to" executes. If none of the guards is true, the alternative statement aborts. The syntax for the repetitive statement is:

\[
\text{do } \text{condition-1 } \to \text{statement-1, ...} \\
[ \]
\text{condition-2 } \to \text{statement-2, ...} \\
\ldots \\
[ \]
\text{condition-n } \to \text{statement-n, ...} \\
\text{od}
\]

This statement executes repetitively until none of the guards has a true value. The execution then moves to the next statement.

Temporal Logic and Operators:: Temporal logic, defined by Pnueli [Pnu77], contains temporal operators which specify time concepts. The general format of temporal logic is very similar to that of propositional logic. Temporal logic formulas are defined by combining temporal operators, logical operators and logical expressions. Use of temporal logic for specifying the behavior of software systems is introduced in [Bar86] and [Lam82]. The basic temporal operators are as follows:

\(\square\) : always true in the future
\(\Diamond\) : the next state is true
\(\lozenge\) : sometimes or eventually true in the future

These temporal operators and temporal logic expressions are used to specify constraints and communication patterns in this language.
2.3 Distributed Object-Oriented Systems

2.3.1 Distributed Systems

Multiple definitions of distributed systems exist. We use the following definition:

A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by messages over communication networks [Bali89].

The physical primitive property of distributed systems is that each processor with its own local memory executes independently. There is neither a global memory nor a central control unit. A processor executes its own instructions with its local memory autonomously and communication between processors is performed by message passing. In general, a distributed system has a network such as hypercube, local-area network, or wide-area network [Bali89]. We can divide distributed system architectures into two types according to type of the communicating network. One is the closely coupled type in which communication is fast and reliable since the processors are very close to each other. The other is the loosely coupled type which has slow and unreliable communication between processors that are located very far each other.

There are many benefits of distributed systems over non-distributed systems. Advantages of distributed systems include decreased turnaround time, increased performance, high reliability and availability, sharing of resources, and achievement of inherent parallelism [Son88], [Bali89]. However, distributed systems introduce other problems, such as deadlock, the unboundedness problem in message queues and message collision [Son88]. As the processors are geographically separated, communication costs must be considered. Moreover, the order of execution of each process cannot be predetermined. Thus, during the design phase of distributed systems,
communication and nondeterministic execution aspects should be emphasized. Despite inherent benefits, the development of software support for distributed systems is typically slower than that of the hardware similar to the general phenomena of software and hardware development.

Many distributed programming languages have been developed for the implementation of distributed systems. Initially, sequential programming languages were used in combination with operating system primitive operations to write distributed computing programs [Bal89]. Since such approaches are generally inadequate, new languages for distributed systems have evolved such as Concurrent Pascal, Concurrent C, CSP and Ada. A specification language of distributed systems is indispensable for system development because behaviors and communication patterns of distributed systems are more complex than non-distributed systems. In a general case, testing and debugging of distributed systems are also much harder than for sequential processing systems. We can help overcome such difficulties by analyzing a specification using formal tools. For example, research on the use of Petri nets for modeling and analyzing distributed systems can be found in [Pet77], [Pet81], [Yau83], [Sha89] and [Sha90].

2.3.2 Distributed Object-Oriented Systems

As object-oriented systems and distributed systems share similar properties, the combination of these systems is somewhat natural. To identify parallelism in a software system, the main objective is to decompose the system into modules that can execute in parallel. An object in an object-oriented system inherently has a suitable form for a distributed system [Yon87]. There are two ways to exploit parallelism in distributed object-oriented systems. One way is to define an object and a process
separately, i.e., one process can contain several object modules simultaneously, and
the other way is to regard an object and a process as the same parallel execution com­
ponent by assigning a process to to each object [Bal89]. The second approach is more
widely used for distributed object-oriented systems because the parallel execution
components, objects, can be easily identified [Yok87]. Most distributed object­
oriented programming languages support parallelism between objects, but a few lan­
guages support inter-object concurrency. Generally, the definition and size of an
object is not standardized so it is different according to the programming languages.

In this work, we assume that only active objects can be activated initially and
execute concurrently, i.e., passive objects can be activated after they receive mes­
sages from active objects. A passive object is an object which merely receives
action(s) from another object or objects, and an active object is an object which per­
forms action(s) on other object(s) or receives actions from another object. Messages
can be passed either synchronously or asynchronously, depending on the implementa­
tion.

Actor, the first computational model for a distributed object-oriented approach,
has become a base model for the development of distributed object-oriented program­
ing languages. Actor is a concurrent computational model in which an actor object
communicates with other actors by message passing [Agh86]. An actor has its own
mail box where messages are queued. The behavior of an actor, called a script, per­
forms actions according to the message sent to it. The actor model provides "inherent
concurrency" which means that the concurrency aspects are expected by the structure
of programs [Agh86]. Representative distributed object-based languages include
ABCL/1 [Yon87], Act/1 [Lie87], SINA [Tri89], and ConcurrentSmalltalk [Yok87].
2.3.2.1 Specification of Distributed Systems

In [Son88], four approaches to specifying distributed systems are introduced: transition-oriented, language-oriented, hybrid, and algebraic. In the transition-oriented approach, a system is represented by states and events. Finite state machine, Petri nets, and IC* [Cam88] are representative models for this group. These models have graphical formats, thus providing relative ease of understanding of executional aspects. In particular, a Petri net representation is widely used to specify the communicational and nondeterministic aspects in distributed systems since such facts can be represented by Petri nets conceptually and graphically. In a language-oriented approach, the computational aspects of a system are represented by formal specification languages or programming languages. Representation of safety and liveness properties of protocols are emphasized. The advantages of this approach include easy implementation, strong modeling power and ease of modeling data transfer. It also offers some disadvantages such as difficulty of modeling logical properties, difficulty of achieving automatic implementation and increased complexity according to the size of a system. A hybrid approach is the combination of the transition-oriented approach and the language-oriented approach. In an algebraic approach, axioms are used to specify the requirements of systems [Win90] and abstract data types and processes are specified in the form of algebras. CCS is a representative model of the algebraic model in which the process is specified in an algebraic pattern [Son88]. While a transition-oriented approach provides easiness of system modeling and understanding, the language-oriented approach enhances the implementation process.
2.4 A Specification Language, DOSL

2.4.1 Motivation behind Development

Even though the use of formal specification languages provides important benefits, they still find limited use in real software development. In some cases, this phenomenon is due to the complexity of the requirements and in other cases due to hesitation of the system designer. Some problems with the use of specification languages include:

i) The formal specification is difficult to understand, particularly for complex systems. Therefore, special features which help the incremental development of specifications are needed.

ii) There is a lack of techniques which help map the specification into implementation. The transferral of a specification into executable code is a difficult task. The syntax and semantics of the specification language often differ greatly from the syntax and semantics of the implementing language. There are two techniques to address this problem. One is to extend a programming language to include specification primitives. Another approach is to specify the specification into two parts: a language-independent part and a language-dependent part. Anna [Luc85] uses the first approach and Larch uses the second approach.

iii) Formal specification languages for object-oriented systems are not as common as are conventional specification languages. Distributed object-oriented specification languages are even less common. Of those object-oriented specification languages that are available, most lack extensive communication features.
2.4.2 Major Features of DOSL

DOSL provides a formal mechanism for representing the initial object specification model. It supports data abstraction and provides features for representing communication aspects. Primary design goals are formalism, message passing, constraints and abstraction as given in Table 2.1. The major features of DOSL are:

a) features of object-oriented systems:

Wegner [Weg87] addresses the characteristics of object-based languages in terms of objects, classes, inheritance, data abstraction, strong typing, concurrency and persistence. DOSL supports an abstract data type by modularizing a system according to objects with their local variables and operations. Inheritance is the property where a child object inherits the data and behavior from its class object(s). The benefits of inheritance include reusability, code sharing and consistency of interface [Bud91]. The inheritance mechanism is different from the object-oriented programming language. In DOSL, a detailed inheritance mechanism is not included. We have included a slot, for class but the full use of inheritance is a future research issue. In most distributed object-oriented languages, parallelism/concurrency is achieved by assigning a process to each object so that objects become active components which execute in parallel. In some cases, a single process can contain multiple object modules simultaneously. In DOSL, we assume that each object module will be assigned to a process.

b) features of distributed systems:

Required features of languages which implement distributed systems are parallelism/concurrency, communication and fault-tolerance [Bal89]. If we
develop distributed systems from an object-oriented perspective, parallelism is achieved because each object is regarded as a computational component which can execute in parallel with other objects. Communication patterns are explicitly defined. The detailed message passing features of DOSL are discussed below.

c) features for message passing:

The methods of message passing between modules are dependent on the underlying communicational mechanism. The actual message passing mechanism is an implementation concern. We assume that each object module has a mailbox which stores incoming messages and performs as a queue. An object in active state checks its mailbox to see whether any message is there. A message consists of an operation name and a set of parameters which can be null. We assume that the sender object's name is attached to the message when a message is stored in a queue so that the receiver object recognizes the object to which to send back the requested information. An object is activated when it receives a message(s) from other object(s). However, the active object is initially activated by a system object which is not specified in the specification model. We assume that there exists a system object in each specification model, which initially sends a message to the method \[\text{new}\] in the active object. This method acts as the constructor.

There are two different methods of communication in distributed systems: synchronous and asynchronous. These two communication patterns are introduced pictorially in Figure 2.1. The temporal operators precede the message
sending statements to specify the communication method. While the two operators □ and ♦ are used to denote the asynchronous message passing, the operator ○ is used to represent the synchronous message passing according to their meaning. The operator □, which means that the statement is always true, is used to denote the asynchronous message passing in which the sender object does not receive any information back from the receiver object, i.e., it continues execution immediately after sending any message. The operator ♦ is used to represent the asynchronous message passing in which the sender object receives the requested information eventually but it does not suspend its execution in the meantime. The operator ○ is used to denote the synchronous communication in which the sender object has to wait until the receiver object gets the message or until it receives the requested information. To simplify the specification, the operators □ and ○ can be omitted when the communication pattern is obvious.

DOSL provides a method to define the priority of a message with a number between 1 and 10, i.e., 1 represents the lowest priority and 10 represents the highest priority. The default priority is 1. The priority number of a message is specified as a superscript of arrow "\(\Rightarrow\)" in the message accepting statement. The arrival of a higher priority message preempts the executing lower priority message. When a message with higher priority is entered during the execution of a method, the lower priority message is suspended and the higher priority method executes. After the execution of higher priority method is completed, the suspended lower priority method resumes. Some distributed object-oriented programming languages such as ABCL/1, PO [Cor90], and Orient84/K [Ish87] support the priority strategy in the message. For messages
in a queue, they are served according to the priority. Messages with the same priority are serviced by a first-come first-service policy.

**d) features for specifying system's constraints:**

While the property-oriented languages emphasize the specification of constraints of a system, model-oriented specification languages stress the internal behavior of a system. Even though DOSL primarily follows the model-oriented approach, it uses the property-oriented to specify the system constraints. Constraints are specified in terms of temporal logic formulas which are appropriate for expressing constraints of distributed systems as well as real-time systems. The reserved word "INIT" represents the initial state of the system. The use of temporal logic as a tool for the specification in distributed systems is discussed in [Ban89].

![Figure 2.1 The two message passing methods](image-url)
2.4.3 Informal Interpretation of a DOSL Specification

A DOSL specification consists of a set of object modules. Each module specification consists of two components: definition and body. The interface of an object is specified in the definition component, and the behavior of an object is specified in the body component. The dynamic behavior and communication aspects of each object are emphasized. The general format of an object module is shown in Figure 2.2 and each component is explained in the following sections.

2.4.3.1 Definition Component

The definition component, which represents the interface of an object, is composed of three parts: the environment, the list of operations, and the constraints of an object. The environment information includes type, class objects, visible objects and local variables of an object, which are explained in Figure 2.2. The operations of an object are represented in terms of signatures, so that the specifier immediately recognizes the input and output items of each method. Such a representation is influenced by the algebraic specification technique which specifies the relationships between objects and operations. The constraints in an object module are specified in terms of linear temporal logic.

2.4.3.2 Body Component

The body component of the object module consists of a set of methods. Each method name is represented by the method name and a set of parameters which can be null. Each method is blocked by \textit{begin} and \textit{end} clauses. Within a block, a set of statements which specifies the behavior of each method is specified. Comments can be described anywhere by using the symbol ":--" at the beginning of the comment.
ObjectModule :: [object]

Definition is

  type      ::= passive or active
  class     ::= parent objects
  visible   ::= visible objects
  variable  ::= the state variables of the object

Methods

  ::= operations of an object are defined in terms of signatures.

Constraints

  ::= constraints of an object are specified

Body is

  (=>^p [::method-name] begin
    statement;
    statement;
    statement;
    
    statement;
  end)

.
.
.

End.

Figure 2.2 Object module specification
2.5 Examples

We give the specification for two well-known problems. Each problem and its DOSL specification are followed by an informal explanation. Statement numbers, which are not part of the official language definition, are added to facilitate the explanation.

2.5.1 Example 1: Producer - Consumer Problem

The classic producer-consumer problem is specified in DOSL to illustrate the synchronous message passing method. Two objects, producer and consumer, continuously put and get items in the buffer under the valid conditions. The DOSL specification of this problem is as follows.

**ObjectModule**: [BoundedBuffer]

**Definition is**

1.1 type : passive
1.2 class :
1.3 visible : [Producer], [Consumer]
1.4 variable : Item; Buffer := array [1..N] of Item;
1.5 I := integer;

**Methods**

1.6 method deposit : Item \rightarrow nil
1.7 method remove : ( ) \rightarrow Item

**Constraints**

1.8 □ ( ~ Buffer(N+1) ∩ ~ Buffer(-1))
1.9 □ (INIT \rightarrow Buffer(0))

**Body is**

1.10 (⇒ [:deposit (Item)] when ~ (Buffer(N))

:: this method can be invoked when the buffer is not full

:: An item is saved to the buffer

1.11 begin
1.12  I := I + 1;
1.13  Buffer(I) := Item;
1.14  end )

1.20  (=> [:remove ] when \( \neg (Buffer(I)) \))
      --- this method can be invoked when the buffer is not empty
      --- An item is removed and sent to the requesting object
1.21  begin
1.22  Item := Buffer(I);
1.23  I := I - 1;
1.24  [ANY] <= [:Item];
1.25  end )
End;

ObjectModule :: [Consumer]

Definition-part is

2.1  type : active
2.2  class :
2.3  visible : [BoundedBuffer]
2.4  variable : Item;

Methods
2.5  method new : ( ) \rightarrow nil

Constraints
\( \emptyset \)

Body-Part is
2.10  (=> [:new] begin
2.11  repeat[
2.12  true \rightarrow o (Item := [BoundedBuffer] <= [:remove ] )
2.13  --- consume the received item
2.14  ] end )

End;
ObjectModule :: [Producer]

Definition is

3.1 type : active
3.2 class :
3.3 visible : [BoundedBuffer]
3.4 variable : Item;

Methods

method new : ( ) —► nil

Constraints

Ø

Body is

3.10 (=> [:new] begin
3.11 repeat[
3.12 ::= produce an item and store it in the buffer
3.13 true —> o ([BoundedBuffer] <= [:deposit (Item) ] ) ;
3.14 | end )

End.

This specification model consists of three objects: [Producer], [Consumer] and [BoundedBuffer]. The two active objects [Producer] and [Consumer], are activated by a system object which sends a message [:new] to them. The message passing within this model is performed synchronously, thus only the operator o is used as a prefix in the message sending statements.

The object [BoundedBuffer] is a passive object that communicates only with [Consumer] and [Producer]. The passive object is invoked only after it receives a message(s) from other object(s). The signatures of the methods are expressed in 1.6-1.7. The priorities of these two messages are omitted because they have the same default priority. A constraint in [BoundedBuffer] is that it does not allow underflow or overflow on the Buffer (1.8). In addition, the Buffer is initially empty (1.9). In the
body of the specification, the internal behavior of two methods is specified. When a message \([\text{deposit}(\text{item})]\) arrives and the Buffer is not full, statements 1.11-1.14 execute. If the Buffer is not full, the incoming data item is stored in the Buffer. If the Buffer is full, the message is suspended until the condition becomes true. When \([\text{remove}]\) message is accepted, the data item on the top of Buffer is removed and sent back to the sender object, unless the Buffer is empty. If there is no data item, it waits until the Buffer is not empty. \([	ext{ANY}]\), in statement 1.24 indicates the object which has sent a message \([\text{remove}]\) to the \([\text{BoundedBuffer}]\). In this statement, the temporal operator \(\diamond\) is omitted because the communication pattern has already been determined by the partner object, the \([\text{Producer}]\), in the statement 3.13.

The object \([\text{Consumer}]\) is an active object which sends a message \([\text{remove}]\) to the \([\text{BoundedBuffer}]\) and expects the variable \(\text{item}\) to receive a data item from the \([\text{BoundedBuffer}]\). The operator \(\diamond\) is used to indicate the synchronous message passing. The execution of the next statement (2.11) needs to be suspended until the \([\text{Consumer}]\) receive a data item. When the \([\text{BoundedBuffer}]\) sends an item, the item will be consumed. This execution repeats forever.

The object \([\text{Producer}]\) produces a data item and sends it to the \([\text{BoundedBuffer}]\) to save it into the Buffer. Since there are no constraints on this object, the symbol \(\emptyset\) is used. This execution repeats forever.

### 2.5.2 Example 2: Problem Solving Organization

This second example adopted from \([\text{Yon87}]\) illustrates the asynchronous message. Assume there is a problem solving team organized to solve a problem within a specified time. The team consists of a manager, a project leader, and several problem solvers. The manager has a problem and submits the problem specification with the
time limit to the project leader. The project leader distributes the problem to each problem solver. He also works on the problem. When a problem solver gets a solution, he sends it immediately to the project leader. From all of the solutions, the project leader selects the best solution and reports it to the manager. After sending the report, he instructs the project solvers to terminate their work on the project. He also stops working on the project. If no solution is found within a given time, the project leader can ask the manager to extend the time. The manager makes the decision whether or not to extend the time. If no solution is found by the end of the time limit, the project leader announces to everyone to stop working on the project and he terminates himself.

From the proposed system, four objects, [Project-Leader], [Manager], [Problem-solver], and [Alarm] are identified. The specification model for this system is given as follows.

**ObjectModule** :: [Manager]

**Definition** is

1.1 type : active
1.2 class :
1.3 visible : [Project-Leader] [Alarm]
1.4 variable : Time, Time-left, Enough := integer; Problem;

**Methods**

1.5 method found : Solution → nil
1.6 method extend-time : () → Boolean
1.7 method no-solution : () → nil
1.8 method time-is-up : () → nil
1.9 method new : () → nil

**Constraints**

ø
Body is

1.10 (=>10 [:new] begin
1.11    [Alarm] <= [:wake-me (Time)];
1.12    [Project-Leader] <= [:start-solving (Time, Problem) ];
1.13 end
   ;;
1.20 (=>10 [:found (Solution)] begin stop end)
   ;;
1.30 (=> [:extend-time ] begin
1.31    ♦ (Time-left := ([Alarm] <= [:how-much-left]));
1.32    if Time-left > Enough then
1.33       ([ANY] <= [:'yes']);
1.34    else
1.35       ([ANY] <= [:'no']);
1.36 end
   ;;
1.40 (=>5 [:time-is-up] begin [Project-Leader] <= [:you-are-late] end)
   ;;
1.50 (=>10 [:no-solution] begin stop end)

End;

ObjectModule :: [Project-Leader]

Definition is

2.1 type : passive
2.2 class :
2.3 visible : [Manager], [Problem-Solver], [Alarm]
2.4 variable : Solution; MySolution; BestSolution;
2.5 N := integer;

Methods
2.6 method start-solving : TimeXProblem → BestSolution
2.7 method time-is-up : () → nil
2.8 method you-are-late : () → nil

Constraints
Ø

Body is

2.10 (=> [:start-solving (T, Problem)] begin
2.11    [Alarm] <= [:wake-me-up (T)];
2.12    ♦ (Solution := ([Problem-Solver] <= [:solve-problem (Problem)]));
while (Solution = null OR MySolution=null) do
    :- solve the problem
    od

    :- find the best solutions from solutions
    [Manger] <= [:found (BestSolution)];
    [Alarm] <= [:stop-your-work];
    [Problem-Solver] <= [:stop-work];
    stop; end )

;

(=\^[time-is-up] begin
    if (Solution=null \cup MySolution = null) then
        ♦ (Response := [Manager] <= [:extend-time ]);
        select{ Response = 'yes' -> [Alarm] <= [:wake-me (20)]
            [Alarm] <= [:stop-your-work];
            [Manager] <= [:no-solution];
        Response = 'no' -> [Problem-Solver] <= [:stop-work];
        [Alarm] <= [:stop-your-work];
    fi end)

;

(=\^[you-are-late] begin
    if (Solution=null) then
        [Alarm] <= [:stop-your-work];
        [Manager] <= [:no-solution];
    fi
    stop; end)

End;

ObjectModule :: [Alarm]

Definition is

3.1 type : passive
3.2 class :
3.3 visible : [Project-Solver] [Manager]
3.4 variable : Time :integer;

Methods
3.5 method wake-me : Time \rightarrow nil;
3.6 method stop : () \rightarrow nil;
3.7 method how-much-left : () \rightarrow Time;

Constraint
\emptyset
Body is

3.10  (=> [:wake-me (Time)] begin
3.11     while Time > 0 do
3.12         Time := Time -1;
3.13     od;
3.14  [ANY] <= [:time-is-up]; end )

;;
3.20  (=> 10 [:stop-your-work] begin stop end )

;:
3.30  (=> 5 [:how-much-left] begin
3.31      [ANY] <= [: (Time)]; end )

End;

ObjectModule :: [Problem-Solver]

Definition is

4.1  type : passive
4.2  class :
4.3  visible : [Project-leader]
4.4  variable : Problem

Methods
4.5  method solve-problem : Problem -> Solution
4.6  method stop-work : () -> nil

Constraints
Ø

Body is

4.10  (=> [:solve-problem (Problem)] begin
4.11      while (Solution = null) do
4.12         :- solve the problem
4.13      od
4.14      if Solution <> null then
4.15         [ANY] <= [: (Solution)]; end )

;:
4.20  (=> 10 [:stop-work] begin stop end )

:- When he is commanded to stop the work, he terminates himself.

End.
The active object [Manager] is activated when a message [:new] is sent from a system object. The communication between object modules is done asynchronously with the two temporal operators □ and ♦ preceding the message sending statements. Initially, an object [Manager] sets up a time limit and commands the object [Project-Leader] to start solving the problem within the given time limit. If the [Manager] receives a message [:found(Solution)], then it terminates. If a message [:extend-time] arrives, he gets the time remaining from [Alarm]. If the time remaining is sufficient, then he sends [:('yes')] to the requested object. Otherwise it sends [:('no')]. If there is a message [:time-is-up], then he announces to the [Project-Leader] a warning message [:your-are-late]. These messages are serviced according to priority number.

The object [Project-Leader] begins to execute when a message [:start-solving (T,Problem)] is sent. Variables T and Problem represent a time limit and the problem description, respectively. The [Project-Leader] lets the [Alarm] know his time limit so that he can try to solve the problem within a given time. He sends a message containing the problem specification to the [Problem-Solver] and expects to receive the solution back. In the meantime, he tries to solve the problem himself. Regularly, he checks a variable Solution to determine whether any solution has been sent to him. When he receives a solution or has his own solution, he sends the best solution to the [Manager] and lets other objects terminate. When a message [:time-up] arrives, the [Problem-Leader] requests the [Manager] to extend the time. If [Manager] sends back a message "yes" then he sends a message to [Alarm] to extend "20" time units. If the [Manager] sends the negative response, it tells [Problem-Solver] to stop the work. If a message [:you-are-late] arrives and there is no solution, then he announces to other objects to stop and he terminates himself.
The passive object [Alarm] counts the time. It decreases the input time until it is zero at which time it sends the message [:time-is-up] to the sender object. When a message [:how-much-left] is sent, the time remaining is sent back to the requested object.

The object [Problem-Solver] can only communicate with the [Problem-leader]. When a message [:solve-problem (Problem)] arrives, he tries to solve the problem until he gets the solution. When he gets the solution, he sends a message back to the sender object. When a message [:stop-work] arrives, he stops working.

2.6 Syntax of DOSL

The formal definition of syntax of DOSL is given below in extended Backus-Naur Form.

\[
\text{<parallel-module>} ::= \text{<dist-module>} \mid \text{<dist-module>} \text{<parallel-module>}
\]

\[
\text{<dist-module>} ::= \text{<module>} \mid \text{<module>} \text{<par-op>} \text{<module>}
\]

\[
\text{<module>} ::= \text{ObjectModule :: <object>}
\quad \text{<definition-section>}
\quad \text{<constraint-section>}
\quad \text{<body-section>}
\quad \text{End}
\]

\[
\text{<definition-section>} ::= \text{Definition is}
\quad \text{type} : \text{<type>}
\quad \text{class} : \text{<object-list>}
\quad \text{visible} : \text{<object-list>}
\quad \text{variable} : \text{<declaration-sequence>}
\quad \text{method} : \text{<method-declaration>}
\]

\[
\text{<type>} ::= \text{active} \mid \text{passive}
\]

\[
\text{<object-list>} ::= \text{<object>} \mid \text{<object>} \text{<object-list>}
\]

\[
\text{<object>} ::= [\text{<identifier>}]}
\]
<declaration-sequence> ::= <declaration>
    | <declaration> <declaration-sequence>

<declaration> ::= <identifier> { := <data-type> } ;
<data-type> ::= integer | real | string | boolean | <array-type>
[array-type] ::= array [integer..integer] of <data-types>

<method-declaration> ::= <method>
    | <method> <method-declaration>

<method> ::= method <identifier> ( ) : <op-sequence> — » <return-value>
<op-sequence> ::= <identifier> ( x <identifier> }
<return-value> ::= nil | <identifier>

<constraint-section> ::= constraints
    <logic-exp-sequence> | ∅

<logic-exp-sequence> ::= <templogic-exp>
    | <templogic-exp> <logic-exp-sequence>
<templogic-exp> ::= { <temp-op> } ⟨logic-expression⟩ ;
<temp-op> ::= □ (always) | ○ (next) | ♦ (eventually) | → (until)
<par-op> ::= ||

<logic-expression> ::= <sexpression> <relational-operator> <sexpression>
<sexpression> ::= <term> | <signed-term> | <additive-expression>
<term> ::= <factor> | <multiplying-expression>
<bracket-expression> ::= ( <expression> )
<factor> ::= <variable> | <string> | <number> | <bracket-expression>
    | <not-expression>
<bracket-expression> ::= ( <expression> )
<not-expression> ::= − <factor>
<relational-operator> ::= = | ≠ | < | > | <= | >=
<string> ::= <letter> | <letter> <string>
<number> ::= <integer> | <real-num>

<signed-term> ::= <sign> <term>

<multiplying-expression> ::= <sexpression> <multiplying-ops> | <sexpression>

<multiplying-ops> ::= * | / | ∩

<additive-expression> ::= <sexpression> <adding-ops> <sexpression>

<adding-ops> ::= + | − | ≝

<sign> ::= + | −

<body-section> ::= Body is

<declaration-method> ::= <method-exp>

<method-exp> ::= (⇒°, > [[:identifier> {(<iden-list>)}]

{when <logic-exp-sequence>};

begin

<statement-sequence>

| <guardcommand-sequence>

end )

(;;)

<n> ::= <digit>

<iden-list> ::= <identifier> | <identifier> iden-list

<statement-sequence> ::= <statement>

| <statement> <statement-sequence>

<statement> ::= skip | abort | stop

| <assignment-statement>

| <communication-statement>

| <if-statement>

| <while-statement>

<assignment-statement> ::= <identifier> := <expression>;

<communication-statement> ::= |

{<temp-op>} {{<identifier> :=} <object> <= [:identifier> {(<iden-list>)}]});

<if-statement> ::= if <condition> then <statement-sequence>

else <statement-sequence> fi

<while-statement> ::= while <condition> do <statement-sequence> od
2.7 Summary

As the need for distributed computing systems rapidly increases, so does the for methods to support their development. Most existing distributed languages do not adequately support the implementation of distributed systems. We have presented a formal specification language DOSL which was developed to specify distributed object-based systems. DOSL contains the desirable features for specifying and developing distributed systems based on an object perspective.
DOSL has a hybrid format which combines the advantages of the property-oriented approach and the model-oriented approach; the definition part follows the algebraic approach and the body part follows the model-oriented approach. The prominent features of DOSL include 1) the message passing statements, 2) priority in the message, 3) a concise and readable format, 4) temporal operators for representing of message passing methods, and 5) support for data abstraction.

Related specification languages are included for the comparison with DOSL. The meaning and use of the DOSL specification is informally given by providing two example specification models. The complete syntax of DOSL has defined in extended Backus-Naur Form. The formal semantics of DOSL is discussed in Chapter 3.
3.1 Introduction to Semantics

A formal language is defined by its syntax and semantics. While the syntax of a language defines the rules which must be observed to write a correct program, the semantics provides the meaning of the program. Defining the semantics of the statements in a language is important because it can prevent the language from being misinterpreted and misused. In addition, it is possible to find hidden inconsistencies and to eliminate ambiguous notions in the language by providing the semantics to the language structures. Any attempt to explain the meaning of program structures informally in terms of a natural language is likely to contain inconsistencies, ambiguities and incompleteness. However, a formal semantics eliminates unclear information by representing program structures with mathematics-based techniques, such as an abstract machine, a predicate calculus or a functional calculus [Ghe87]. Therefore, formal semantics are preferred over informal semantics. However, generally both informal and formal semantics are needed to provide a formal, yet pragmatic, description of a language.

There are three main approaches to describe the formal semantics of languages: operational, denotational and axiomatic. The meaning of program structures is explained in terms of changes of state in the program by demonstrating their execution on a hypothetical machine in an operational semantics approach; in terms of mathematical entities in the denotational semantics approach; and in terms of axioms and inference rules in the axiomatic semantic approach. These three approaches are introduced briefly below.
Denotational Semantics:

The denotational semantics approach, introduced by Strachey and Scott [Sto77], has received wide application for the definition of formal languages. In contrast to other two approaches, it is more difficult to understand since language constructs are defined in terms of mathematical objects, such as functions and domains. The meaning of program structures is interpreted in terms of functions by emphasizing input/output behavior instead of providing intermediate states during execution [Man86]. In other words, the meaning of program constructs is given by assigning mathematical values to them [Ame90]. The function which generates a state vector has two arguments; the program construct and a state vector [Don76]. This function defines a state-to-state mapping for the given program of a language. The format of a function is:

\[ \zeta: \text{Program} \rightarrow \text{State} \rightarrow \text{State}. \]

In a sequential program, it is straightforward to define such a function, as the change of states before execution and after execution is clear; however, defining a proper function which can specify the behavior of parallel execution is difficult. The difficulties of denoting a parallel execution function are discussed in [Ame90].

Axiomatic Semantics:

In the axiomatic semantics approach, the meaning of program structures is explained in terms of the state of computation in a mathematical logic expression on program variables [Man86]. Such logical expressions are called axioms. The program construct is explained with axioms and inference rules, i.e., by representing how the construct (statement) changes after execution under the
assumption of a true condition. This approach is widely used also to prove proper-
ties of a program [Mar85]. The basic format of an axiom is:

\[ Pr_1 \ (Stat) \ Pr_2. \]

where \( Pr_1 \) and \( Pr_2 \) are propositions, and \( Stat \) represents a statement. The mean-
ing of this format is: if \( Pr_1 \) is true before a statement \( Stat \) executes, \( Pr_2 \) becomes
true after \( Stat \) executes.

Operational Semantics::

In an operational semantics approach, a program is executed on an abstract
machine and all intermediate states during the execution are traced. The meaning
of a program is defined in terms of state in a program. In other words, the mean-
ing of a program is given by executing each statement with input data on an
abstract machine. Various abstract machines have been used to provide opera-
tional semantics for programming languages. For example, a stack machine with
VDL notation has been used for PL/1 definition [Mar85], a nondeterministic
automaton or a transition system has been used for CCS and CSP [Plo81],
[Plo82], and Petri nets have been used for CSP [Gol84], CCS [Nie87], CCSP
[Old87b], and COSY [Bes87]. Among those abstract machines, Petri nets are an
appropriate machine for providing the operational semantics of concurrent/distributed programming languages. Moreover, Petri nets can represent the
concurrent execution both conceptually and graphically [Gol84]. Jensen wrote a
PASCAL semantics by combining denotational semantics and high-level Petri
nets (operational semantics) [Jen85]. Plotkin introduced his own style of an oper-
tional semantics method by defining transition rules on a transition system
[Hen81], [Plo81], [Plo82]. Although Plotkin’s transition system does not
explicitly define an abstract machine, it shows the details of execution of program structures with the state transitions.

Among the three approaches, an operational semantics approach is regarded as the most intuitive approach as it emphasizes the implementation aspects of a language instead of the descriptive aspects. The main advantage of the operational semantics is that it provides the meaning of program structures by illustrating all possible transitions and eliminating all illegal transitions [Liu89]. The DOSL semantics is given with the operational approach in Section 3.3.

3.2 Abstract Syntax

An abstract syntax which represents the syntactically simplified form of a language plays an important role in the formal definition of a language. An abstract syntax, sometimes called the semantic structure, is derived by eliminating semantically unnecessary notations from the fully descriptive syntax, called the concrete syntax. Thus, the number of rules required for language definition definitely is greatly reduced when an abstract syntax is used instead of a concrete syntax.

For the abstract syntax of DOSL, we use the following syntactic categories (sets).

- ObjectModule $\rightarrow$ the set of object modules, ranged over by $om$;
- Command $\rightarrow$ the set of commands, ranged over by $c$;
- Pcommand $\rightarrow$ the set of parallel commands, ranged over by $pc$;
- OBname $\rightarrow$ the set of object labels, ranged over by $O, P, Q$;
- Bexp $\rightarrow$ the set of boolean expressions, ranged over by $a, b$;
- Aexp $\rightarrow$ the set of expressions, ranged over by $d, e$;
- Msg $\rightarrow$ the set of messages, ranged over by $m, n$;
Var $\rightarrow$ the set of variables, ranged over by $x, y, z$;

Val $\rightarrow$ the set of values, ranged over by $v$;

Temp $\rightarrow$ the set of temporal operators, ranged over by $t$.

An object module $om$ consists of an object label (name) $O$ and a set of commands $c$. A command $c$ consists of various statements. DOSL supports the inter-parallelism between object modules. The parallel command $pc$ is defined by the symbol, 'II'. The abstract syntax of DOSL is defined in a BNF-like format as follows.

\[
om ::= O :: c \\
c ::= \text{skip} | \text{stop} | \text{abort} | x := e | c_1 ; c_2 \\
| \text{if } b \text{ then } c_1 \text{ else } c_2 | \text{while } b \text{ do } c \text{ od} \\
| \text{select}(b_1 \rightarrow c_1 \text{ or } \ldots \text{ or } b_n \rightarrow c_n) \\
| \text{repeat}(b_1 \rightarrow c_1 \text{ or } \ldots \text{ or } b_n \rightarrow c_n) \\
| \text{accept } m(x) \text{ when } b | t(\text{send } n(e) \text{ to } O \text{ & get } y) \\
\]

\[
pc ::= O :: c_1 \ll P :: c_2
\]

We have modified the DOSL abstract syntax so that it can be easily understood instead of using the verbose original clauses or symbols used in the concrete syntax. For example, the message passing statements "$\Rightarrow [m(x)] \text{ when } b$" and "$y := O \Leftarrow [n(e)]$" are modified as "accept $m(x)$ when $b$" and "send $n(e)$ to $O$ & get $y$", respectively. The explanation of each statement will not be given further as statements in the abstract syntax are understandable without further description.
3.3 Operational Semantics of DOSL

The operational semantics of DOSL is defined using two related methods: a transition system and Petri nets. These two methods are not independent on each other. First, a transition system is defined and a set of transition rules on the categories of the DOSL abstract syntax are derived. A Petri net semantics is then derived based on this transition system; that is, the Petri net semantics is defined by representing each transition rule of the transition system in a graphical format. Petri nets are an appropriate tool to generalize the transition system, since they can explicitly represent concurrency and independence between transitions [Deg88]. The firing sequences of Petri nets have been shown to be identical to a transition system method [Gol84]. Therefore, the Petri nets translation of the DOSL abstract syntax categories provides the operational semantics of DOSL. The two methods are discussed separately in Sections 3.3.1 and 3.3.2.

Several definitions are introduced before the operational semantics is given. The State is the values stored in all variables (identifiers) of a program. State in this operational semantics is denoted by a vector which contains a set of defined variables and their corresponding stored values, i.e., \(<(x_1:v_1), (x_2:v_2), \ldots, (x_n:v_n)>\). The initial state of a program is that all variables have no defined values. A change of value of any variable results in the change of state, i.e., state of a program changes according to the change of values stored in variables.

The execution of a program is interpreted as the change of values in a set of defined variables in a program which consists of a set of statements. Particularly, the change of value(s) in the variables is done by an assignment statement or a message passing statement. Other statements control the execution of a program. The
The semantics of a program is denoted with respect to the sequence of states, such as
\[ \sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \sigma_i \rightarrow \ldots \sigma_n, \]
where \( \sigma \) means a normal state. There are three different types of state sequences. First, a program may abort, resulting in an abort state. In this case, the final state \( \sigma_n \) becomes an abort state, \( w \). For example, evaluation of an expression which contains any variable for which no value has been assigned, results in an abort state [Man86].

Second, a finite number of states which means the correct termination of a program. In this case, the range of \( n \) is between 1 and \( \infty \). Third, an infinite number of state sequence means the program runs infinitely. Several definitions concerning the state are introduced below.

**Definition 1**: Given a normal state \( \sigma \), \( \sigma(e) \) is a function which returns the value of an expression \( e \) by evaluating it under the state \( \sigma \). The symbol \( \perp \) represents 'not defined yet' [Man86]. Thus,

1) The evaluation of an expression under a normal state \( \sigma \) is defined by induction as follows.

**Basis step**:  
if \( e \) is an integer, \( \sigma(e) \) results in a number.
if \( e \) is a non-numeric constant, \( \sigma(e) \) results in a constant.
if \( e \) is an identifier (variable), \( \sigma(e) \) results in a value.

**Inductive step**:  
if either \( \sigma(d) = \perp \) or \( \sigma(e) = \perp \), then  
\[ \sigma(d + e) = \sigma(d - e) = \sigma(d * e) = \sigma(d/e) = \perp \]
else \( \sigma(d + e) = \sigma(d) + \sigma(e) \)
\[ \sigma(d - e) = \sigma(d) - \sigma(e) \]
\[\sigma(d \ast e) = \sigma(d) \ast \sigma(e)\]
\[\sigma(d/e) = \sigma(d) / \sigma(e)\]

2) The evaluation of a boolean expression under a normal state \(\sigma\) is defined as follows. Assume that \(T\) and \(F\) represent \textit{true} and \textit{false}, respectively.

\textbf{Basis step:}

\[\sigma(T) = T, \ \sigma(F) = F.\]
\[\sigma(a = b) = \bot\text{ if either }\sigma(a)\text{ or }\sigma(b)\text{ is }\bot, \text{ else is }T\text{ if }\sigma(a)=\sigma(b), \text{ and }F\text{ if }\sigma(a) \neq \sigma(b).\]

\[\sigma(a \neq b), \ \sigma(a < b), \ \sigma(a > b), \ \sigma(a \geq b)\text{ and }\sigma(a \leq b)\text{ are defined in a similar manner.}\]

\textbf{Inductive Step:}

Let \(\neg\) (NOT), \(\cup\) (OR), and \(\cap\) (AND) have their usual meanings on the boolean truth values, \(T\) and \(F\) such as \(\neg T = F, \ \neg F = T, \text{ and } F \cup T = T\), then

\[\sigma(\neg b)\text{ is }\bot\text{ if }\sigma(b)\text{ is }\bot, \text{ else is }\neg(\sigma(b)).\]

\[\sigma(a \cup b)\text{ is }\bot\text{ if either of }\sigma(a)\text{ or }\sigma(b)\text{ is }\bot, \text{ else is }\sigma(a) \cup \sigma(b).\]

\[\sigma(a \cap b)\text{ is }\bot\text{ if either of }\sigma(a)\text{ or }\sigma(b)\text{ is }\bot, \text{ else is }\sigma(a) \cap \sigma(b).\]

\textit{Definition 2.1 ::} \(\sigma[x/v]\) denotes a state which is identical to \(\sigma\) except at a variable \(x\) where a new value \(v\) is assigned. It implies that the value \(v\) is assigned to the variable \(x\).

\textit{Definition 2.2 ::} \(\sigma[x/e]\) means the replacement of the value of a variable \(x\) by the value of expression \(e\) under the same state \(\sigma\). It is an abbreviated notation of \(\sigma[x/\sigma(e)]\).
Definition 3 :: $\sigma'$ means the changed state of the current state $\sigma$ after some values are changed. However, the exact change is unknown.

Definition 4 :: $<c, \sigma>$ means the state sequence of computation in a command $c$ starting from any normal state $\sigma$. Let $w$ mean an abort state. The state sequence is one of the following formats:

- $\sigma, \sigma_1, \sigma_2, ... , \sigma_n, w$ --- the computation aborts
- $\sigma, \sigma_1, \sigma_2, ... , \sigma_n, ...$ --- the computation does not terminate
- $\sigma, \sigma_1, \sigma_2, ... , \sigma_n$ --- the computation terminates

Definition 5 :: $<c, \sigma> \rightarrow^k <c', \sigma'>$ means that the execution of a command $c$ in a state $\sigma$ takes $k$ time units to be in a state $\sigma'$ resulting in a command $c'$. The time unit $k$ equals to 1 when the execution is done within a module with one step transition or one-way communication between two modules. If there is a two way communication between modules, $k$ is greater than 1. When $k$ is greater than 1, the value of $k$ is specified on the transition arrow. Otherwise it is omitted.

3.3.1 The Transition System Method

A transition system is defined and a set of transition rules to denote the change of state before and after the execution are derived for each of the syntactical categories.

Definition 6 :: A transition system has three elements, $(\Gamma, T, \rightarrow)$, where $\Gamma$ is the set of states and $T \subseteq \Gamma$ is the set of terminal states and $\rightarrow \subseteq \Gamma \times \Gamma$ represents the transition relation [Plo82].
For example, a transition system \((\Gamma_c, T_c, \rightarrow_c)\) for a command \(c\) is defined as follows.

\[
\Gamma_c = \{<c, \sigma>\} \cup \{\sigma\} \cup \{w\}
\]

\[
T_c = \{\sigma\} \cup \{w\}
\]

The transition rules for each DOSL statement follow. The expression on the left-hand side of an arrow "\(\rightarrow\)" denotes the condition of the execution, and the right-hand side expression indicates the state transition from pre-state to post-state under the given condition. Sometimes, no condition is needed for the execution of particular statements. "TRUE" is used to represent that case. In the following section, we define the transition relation for each of the commands in DOSL.

**Commands**

* skip, abort and stop

i) TRUE \(\rightarrow\) \(<\text{skip}, \sigma> \rightarrow \sigma\)

ii) TRUE \(\rightarrow\) \(<\text{abort}, \sigma> \rightarrow w\)

iii) TRUE \(\rightarrow\) \(<\text{stop}, \sigma> \rightarrow \sigma\)

The execution of a statement *skip* does not affect the change of a program state. The state remains the same after the execution of *skip*. The statement *abort* results in an abort state \(w\), so that the program stops executing. It interrupts the program execution. The statement *stop* makes the program terminate without changing the state of the program like *skip*. 
• Assignment \([x := e]\)

i) \(\sigma(e) \neq \bot \Rightarrow <x := e, \sigma> \rightarrow \sigma[x/e]\)

ii) \(\sigma(e) = \bot \Rightarrow <x := e, \sigma> \rightarrow w\)

The execution of an assignment statement evaluates an expression \(e\) and assigns a value to the variable \(x\). A state \(\sigma\) changes to \(\sigma[x/e]\) when the evaluation of \(e\) results in a valid value. If the evaluation of \(e\) is undefined, the execution of an assignment statement aborts according to rule ii).

• Sequential composition \([c_1;c_2]\)

i) \(<c_1, \sigma> \rightarrow <c'_1, \sigma'> \Rightarrow <c_1;c_2, \sigma> \rightarrow <c'_1;c_2, \sigma'>\)

ii) \(<c_1, \sigma> \Rightarrow \sigma' \Rightarrow <c_1;c_2, \sigma> \Rightarrow \sigma' <c_2, \sigma'>, \ r > 1\)

iii) \(<c_1, \sigma> \Rightarrow \sigma' w \Rightarrow <c_1;c_2, \sigma> \Rightarrow \sigma' w, \ s > 1\)

iv) TRUE \(\Rightarrow <\text{skip};c, \sigma> \rightarrow <c, \sigma>\)

The sequential composition means the sequence execution of a set of statements, i.e., the first command \(c_1\) executes first, and the second command \(c_2\) follows. If the first command consists of a set of statements, the execution pattern is explained recursively with rule i). Rule ii) says that if the execution of the first command \(c_1\) has completed, then \(c_2\) executes under the changed state \(\sigma'\). According to rule iii), if \(c_1\) aborts, the overall command aborts, irrespective of the second command. The command \(\text{skip}\) does not change the state. The \(r\) and \(s\) represent the time unit used for the execution of the command \(c_1\).
• conditional [if $b$ then $c_1$ else $c_2$]

i) $\sigma(b) = \text{true} \implies \langle \text{if } b \text{ then } c_1 \text{ else } c_2, \varnothing \rangle \rightarrow \langle c_1, \varnothing \rangle$

ii) $\sigma(b) = \text{false} \implies \langle \text{if } b \text{ then } c_1 \text{ else } c_2, \varnothing \rangle \rightarrow \langle c_2, \varnothing \rangle$

iii) $\sigma(b) = \bot \implies \langle \text{if } b \text{ then } c_1 \text{ else } c_2, \varnothing \rangle \rightarrow w$

If a boolean expression $b$ has a true value, then $c_1$ executes. Otherwise, $c_2$ executes. The evaluation of a boolean expression does not change the state. If $b$ is undefined, the statement aborts.

• repetitive [while $b$ do $c$]

i) $\sigma(b) = \text{true} \implies \langle \text{while } b \text{ do } c, \varnothing \rangle \rightarrow \langle c; (\text{while } b \text{ do } c), \varnothing \rangle$

ii) $\sigma(b) = \text{false} \implies \langle \text{while } b \text{ do } c, \varnothing \rangle \rightarrow \langle \text{skip}, \varnothing \rangle$

iii) $\sigma(b) = \bot \implies \langle \text{while } b \text{ do } c, \varnothing \rangle \rightarrow w$

The execution pattern of a while-statement is similar to the guarded iterative statement except that there is only one condition $b$. If $b$ is true, the body statement $c$ executes and the while-statement executes repeatedly until $b$ is false. If $b$ is false, the execution skips to the next statement according to rule ii). If $b$ is undefined, this statement aborts.
• guarded alternative \[\text{select}(b_1 \rightarrow c_1 \text{ or } ... \text{ or } b_n \rightarrow c_n)\]

i) any \(i \sigma(b_i) = \text{true} \implies \langle \text{select} ( ... \ b_i \rightarrow c_i ... ) , \sigma \rangle \rightarrow <c_i,\sigma>

ii) for all \(i \sigma(b_i) = \text{false} \implies \langle \text{select} ( ... \ b_i \rightarrow c_i ... ) , \sigma \rangle \rightarrow w

If more than one guard is true, any one of \(b_i\) which has a true value is chosen nondeterministically and the following statement \(c_i\) executes. If none of \(b_i\) is true, the alternative statement aborts. The evaluation of a boolean expression \(b\) does not change the state.

• guarded iterative \[\text{repeat}(b_1 \rightarrow c_1 \text{ or } ... \text{ or } b_n \rightarrow c_n)\]

i) any \(i \sigma(b_i) = \text{true} \implies \langle \text{repeat}( ... \ b_i \rightarrow c_i ... ) , \sigma \rangle \rightarrow <c_i;\text{repeat} ( ... ), \sigma>

ii) for all \(i \sigma(b_i) = \text{false} \implies \langle \text{repeat}( ... \ b_i \rightarrow c_i ... ) , \sigma \rangle \rightarrow <\text{skip},\sigma>

The execution of the guarded iterative statement continues until all guards are false. If any guard \(b_i\) is true, the corresponding statement \(c_i\) executes like the guarded alternative statement. If more than one condition is true, then a condition is chosen nondeterministically. The behavior repeats until all conditions are false. If all guards are false, the execution proceeds to the next statement without aborting.

• message accepting \[\text{accept} m(x) \text{ when } b\]

i) \(\sigma(x) = \bot \text{ and } \sigma(b) = \text{true} \implies <\text{accept} m(x) \text{ when } b,\sigma> \rightarrow \sigma

ii) \(\sigma(x) \neq \bot \text{ and } \sigma(b) = \text{true} \implies <\text{accept} m(x) \text{ when } b,\sigma> \rightarrow \sigma[x/\nu]
An object module can receive a message from other objects through the message accepting statement. The message consists of an operation name \( m \) and a set of optional variables, called parameters. Even though multiple numbers of parameters may be sent, we include here only one parameter \( x \). If a message contains a parameter(s), the state of the receiver object is changed by assigning the sent value(s) to the parameter(s). It is difficult to explicitly denote the sent value \( v \) without specifying the corresponding message sending statement. Here, we assume that there exists an object which has a message sending statement like "send \( m(v) \ldots \)". The behavior of this statement is similar to that of an assignment statement. If no parameter is sent, this statement does not affect the change of state according to rule i). If there is a parameter(s), the sent value is assigned to the parameter \( x \) as rule ii). These two rules are valid when condition \( h \) is satisfied.

* message sending \( \text{[send } n(e) \text{ to } O \{\& \text{ get } x\}] \) where \( \{\& \text{ get } x\} \) is optional

-. synchronous communication

i) \( \sigma(e) \neq \bot \implies <O(\text{send } n(e) \text{ to } O), \sigma> \rightarrow^{h1} \sigma, h1 \geq 2 \)

ii) \( \sigma(e) \neq \bot \implies <O(\text{send } n(e) \text{ to } O \& \text{ get } y), \sigma> \rightarrow^{h2} \sigma[y/v], h2 \geq 2 \)

-. asynchronous communication

iii) \( \sigma(e) \neq \bot \implies <\Box(\text{send } n(e) \text{ to } O), \sigma> \rightarrow \sigma \)

iv) \( \sigma(e) \neq \bot \implies <\Diamond(\text{send } n(e) \text{ to } O \& \text{ get } z), \sigma> \rightarrow^{k} \sigma[z/v], k > 2 \)
There are two different patterns of message passing in DOSL. The execution of these statements is valid only when the sending expression $e$ has a value. Rules i) and ii) represent the synchronous message sending statement in which a sender object must wait until a receiver object sends back an acknowledgement or the requested information. There is no change of the state in rule i) because the statement only receives an acknowledgement which does not affect the state change after sending a message $n(e)$ to an object $O$. However, when the called object $O$ sends back some information, it changes the state by assigning the information, we also assume a value $v$, into variable $y$ as in rule ii). The time unit for synchronous message passing is greater than or equal to 2.

Rule iii) is defined for an asynchronous message sending statement in which the state remains unchanged since it just sends (outputs) a message to another object $O$ without receiving any information in return. The time unit of this statement is 1. Rule iv) illustrates another asynchronous message sending statement in which the sender does not suspend the execution until some information is sent, but it will eventually receive some information into $z$ from the called object $O$. Like rule ii), the execution of this statement changes the state of a program. In fact, the statements in rule ii) and iv) are a combination of a message sending statement and a message accepting statement. An informal interpretation of the statement in iv) is that "an anonymous object sends a message $n(e)$ to another object $O$, and eventually receives a message from $O$. The received information $v$ is assigned to the variable $z."
* Parallel command \([O::c_1 \| P::c_2]\)

i) \(\langle c_1, \sigma_1 \rangle \rightarrow \langle c_1^{'}, \sigma_1^{'}, \sigma_2 \rangle \rightarrow \langle c_1 \| c_2 \| \sigma_2 \rangle \rightarrow \langle c_1^{'}, \sigma_2 \rangle\)

ii) \(\langle c_1, \sigma_1 \rangle \rightarrow^{p_1} \langle \sigma_1^{'}, \sigma_2 \rangle \rightarrow^{p_1} \langle c_1 \| c_2 \| \sigma_2 \rangle, \ p_1 > 1\)

iii) \(\langle c_2, \sigma_2 \rangle \rightarrow \langle c_2^{'}, \sigma_2^{'}, \sigma_1 \rangle \rightarrow \langle c_1 \| c_2 \| \sigma_1 \rangle \rightarrow \langle c_1 \| c_2 \| \sigma_2 \rangle\)

iv) \(\langle c_2, \sigma \rangle \rightarrow^{p_2} \langle \sigma_2^{'}, \sigma_1 \rangle \rightarrow^{p_2} \langle c_1 \| c_2 \| \sigma_1 \rangle, \ p_2 > 1\)

When two objects \(O\) and \(P\) execute in parallel, their commands, \(c_1\) and \(c_2\), execute in parallel. Assume that \(O\) and \(P\) have different states \(\sigma_1\) and \(\sigma_2\), respectively. In addition, we assume that there is no communication between \(O\) and \(P\). When \(c_1\) executes first, only \(\sigma_1\) changes to \(\sigma_1^{'}, \sigma_2\) without affecting \(\sigma_2\). When \(c_2\) executes, \(\sigma_2\) is changed, vice versa. The order of execution between these two objects is not determined.

The operational semantics of DOSL has been defined in terms of the transition relation which shows the change of state and the remaining program structure to be executed. Basically, the state of a program is changed when it meets an assignment statement or a message passing statement. The semantics of an entire program is denoted by a sequence of states which represents the execution of behavior under the defined transition system rules. This Plotkin-style semantics method was used to define the semantics of the DOSL syntactical categories. In addition, this method shows the detailed executions and state transition steps clearly without running a program on an abstract machine. This Plotkin-style semantics is used as a basis to define Petri net semantics for DOSL. The primary reason for defining the Petri net semantics
is to give a graphical representation that more explicitly shows the meaning of the distributed features of DOSL.

3.3.2 The Petri Net Semantics

A Petri net is defined to represent the transition rules defined in the previous section. The token in the net indicates the program state [Gol84j. In other words, the state of a program is represented as the place where a token appears. We represent intermediate states of a DOSL specification by allowing states $\sigma : Var \rightarrow Val$ as tokens in places. Thus, the sequence of states is explained in terms of a change to the token in the net.

Definition 7:: A Petri net is a triple

$$N = (S, T, F)$$

where

i) $S$ and $T$ are disjoint sets of places and transitions, respectively,

ii) $F \subseteq (S \times T) \cup (T \times S)$ is a relation between places and transitions.

We assume that $s_i$ and $t_j$ represent the elements of two sets, $S$ and $T$. $S$, $T$ and $F$ are represented by circles, boxes and arcs in the net, respectively. In this net representation, the place represents the state of a program and the transition represents the condition which requires to be satisfied for transition of the state.

The firing rule of the Petri net is as follows.

1) a transition $t_i$ can only fire when all incoming places have tokens and the token moves to the outcoming place(s).
2) if a place is connected to more than one outcoming transition, the token can move to a non-deterministically chosen place.

3) if more than one transition enables to fire in the net, any of transitions can fire.

4) a tuple \((s_i, s_j, t_k, s_m, s_n)\), used in this chapter, means that if both of the incoming places \(s_i\) and \(s_j\) have the tokens, the transition \(t_k\) fires and consequently both of the outcoming places \(s_m\) and \(s_n\) have the tokens.

The Petri net semantics provides an advantage over the transition system method, i.e., it enables the analysis of a DOSL specification as well as definition of its semantics in terms of firing sequences in the nets. In addition, the Petri net representation illustrates the execution pattern of each statement graphically and dynamically.

A net transition function \(\eta\) is defined by giving the abstract syntax categories and the corresponding Petri net representations. In order to simplify the net, some net representations do not include the abort state. The statements and the corresponding net representations are defined as follows.

- **skip**

\[
\eta[\text{skip}] := \begin{array}{c}
\end{array}
\]

The statement *skip* does no action, so the state does not change.

- **abort**

\[
\eta[\text{abort}] := \begin{array}{c}
\end{array}
\]
The statement *abort* makes the program terminate abnormally, represented as an infinite loop in the net. However, it is different from an infinite execution of a program.

- **stop**

\[ \eta[\text{stop}] := \quad \]

This statement simply makes the program terminate normally. There is no change of the state; the token does not move.

- **assignment**

\[ \eta[x := e] := \quad \]

The assignment statement is an atomic statement which causes a change of the state. When an initial place \( s_1 \) has a token \( \sigma \) and the expression \( e \) is valid, \( t_1 \) fires and it yields a new marking of the net, i.e., the token is removed from \( s_3 \) to \( s_2 \) which represents the state \( \sigma[x/e] \). When the evaluation of an expression \( e \) results in undefined, the statement aborts.

- **sequential composition**

\[ \eta[c_1;c_2] := \eta[c_1] \circ \eta[c_2] := \quad \]
The sequential composition consists of a set of commands which execute in sequential order. It can be decomposed by statements and be applied by transition function to each component separately.

* conditional

\[
\eta [\text{if } b \text{ then } c_1 \text{ else } c_2] := \]

If condition \( b \) is true, the transition \( t_1 \) fires and the token \( \sigma \) in \( s_1 \) moves to \( s_2 \). If \( b \) is not true, \( t_2 \) fires and the token moves to \( s_2 \). Otherwise, the token reaches an abort state.

* repetitive

\[
\eta [\text{while } b \text{ do } c] := \]

While \( b \) is true, the transitions \( t_1 \) fires and \( t_2 \) fires consequently until \( b \) is false. During this repetition, the state might be changed depending on the statement \( c \). If \( b \) is false, the execution skips to the next statement by firing \( t_3 \).
The execution pattern of the guard-alternative is very similar to that of the if-then-else statement except there is more than one condition (called the guard). One of the guards which has a true value is selected nondeterministically and corresponding $t_i$ fires. Thus the token $\sigma$ moves from $s_1$ to $s_i$. If none of the guard is true, $t_n$ fires and this statement aborts.

If any guard $b_i$ has a true value, the token moves from the initial place $s_1$ to $s_i$ and repeats such a transition until all guards are false. If all guards are false, $t_n$ fires and the execution precedes to the next statement without aborting.
message passing statement

In the Petri net semantics, the message passing statements are specified as a combination of a message sending statement and a message accepting statement. Two different patterns of message passing methods are specified separately. Since the communication is performed between two different modules, each net has two different states; \( \sigma_1 \) is for the sender object (client) \( O_1 \) and \( \sigma_2 \) is for the receiver object (server) \( O_2 \). The intermediate transitions of execution which are not involved in the communication, are abbreviated and denoted by the symbol "".

\[
\eta \left[ (send \: n(e) \: to \: O_2 \: \& \: get \: y) \right] + \eta \left[ accept \: m(x) \right]
\]

The meaning of synchronous message passing is represented a Petri net and given in Figure 3.1. Assume that initially two places \( s_1 \) and \( s_3 \) have the tokens \( \sigma_1 \) and \( \sigma_2 \), respectively. When \( O_1 \) sends a message, \( t_1 \) fires and \( s_2 \) takes the token. Then, \( t_2 \) enables to fire because \( s_2 \) and \( s_3 \) have the tokens, that is, \( O_2 \) receives the message in the queue and continues execution. Meanwhile, \( O_1 \) waits until it receives back the information or an acknowledgement: \( t_3 \) can only fire when the two places \( s_4 \) and \( s_5 \) have the tokens, that is, when \( O_2 \) sends back the information or an acknowledgement to \( O_1 \), \( O_1 \) resumes the execution by firing \( t_3 \).

\[
\eta \left[ (send \: n(e) \: to \: O_2) \right] + \eta \left[ accept \: m(x) \right]
\]

Asynchronous message passing which does not receive back any information from the receiver object is represented in Figure 3.2. Initially, the places \( s_1 \) and \( s_3 \) have the tokens which represent the states of two objects, \( \sigma_1 \) and \( \sigma_2 \).
respectively. When $O_1$ sends a message to $O_2$, the transition $t_1$ fires and consequently $t_2$ fires: $O_2$ has received a message from $O_1$. After that $O_1$ and $O_2$ execute in parallel independently.

$$\eta \cdot (\text{send } n(e) \text{ to } O_2 \& \text{get } y) + \eta \text{[accept } m(x)]$$

The meaning of asynchronous message passing which requires to receive back the information from the receiver object is given in Figure 3.3. The execution pattern of this statement is very similar to that of the synchronous message passing except that the sender object $O_1$ does not need to be suspended until it receives back a information. However, $O_1$ eventually receives back the information from $O_1$, that is, the token in $s_7$ can be moved to any of the places $s_4$, $s_6$, or $s_9$ at some time.

![Figure 3.1 The synchronous message passing](image)
Figure 3.2 The asynchronous message passing without returning information

Figure 3.3 The asynchronous message passing with returning information
Parallel commands

When two objects execute in parallel, it explicitly means that two commands $c_1$ and $c_2$ execute in parallel. The execution pattern in Petri nets is simple. We assume that there is no interaction between two objects.

The Petri net representation for each abstract syntax category is defined in an inductive manner. For translating of a DOSL specification into a Petri net representation, there are two steps. First, each statement in a specification is translated into the corresponding net according to the function $\eta$. Second, the set of nets is combined sequentially according to the order of the statements in a specification. The composition rule is as follows. Suppose two nets $n_1$ and $n_2$ are joined sequentially. The last place of $n_1$ is joined with the first place of $n_2$. Initially, a token is placed on the first place of the combined net. The token will move through the net according to the execution of the statements. From an overall view, the semantics of a program is interpreted in terms of the sequence of the token firings. In addition, the internal behavior of each object module can be analyzed with the firing of a token in the Petri net.

3.4 Examples

We illustrate the elaboration of a simple DOSL specification and its operational semantics with the transition method and Petri nets. We then give a more complete example showing the message passing statements. The first specification is given in the following page.
begin
  \( x := 1; \quad y := 2; \quad (c_1) \)
  \( x := y + 3; \quad (c_2) \)
  \text{if } x > y \text{ then } x := y \text{ else } y := x; \quad (c_3) \)
end.

The transition sequences are:

\[
\sigma_0 = \langle (x:1),(y:1) \rangle \quad \text{(before execution)}
\]
\[
\langle (c_1;c_2;c_3),\sigma_0 \rangle \rightarrow \langle (y := 2;c_2;c_3),\sigma_1 \rangle \quad \text{(by sequential composition)}
\]
\[
\langle (y := 2;c_2;c_3),\sigma_1 \rangle \rightarrow \langle (c_2;c_3),\sigma_2 \rangle \quad \text{(by assignment)}
\]
\[
\langle (c_2;c_3),\sigma_2 \rangle \rightarrow \langle c_3,\sigma_3 \rangle \quad \text{(by assignment)}
\]
\[
\langle c_3,\sigma_3 \rangle \rightarrow \langle x := y,\sigma_3 \rangle \quad \text{(by if-then-else)}
\]
\[
\langle x := y,\sigma_3 \rangle \rightarrow \sigma_4 \quad \text{(by assignment)}
\]

where \( \sigma_0 = \langle (x:1),(y:1) \rangle \) (the initial state)

\[
\sigma_1 = \langle (x:1),(y:1) \rangle
\]
\[
\sigma_2 = \langle (x:1),(y:2) \rangle
\]
\[
\sigma_3 = \langle (x:5),(y:2) \rangle
\]
\[
\sigma_4 = \langle (x:2),(y:2) \rangle \text{ (the final state)}
\]

The specification is valid because the number of sequence of states is finite and it does not reach an abort state.

The Petri nets representation of this specification is shown in Figure 3.4. The Petri nets representation is built using the function \( \eta \) as defined in Section 3.3.2.
Initially, a token is placed at the beginning of the Petri net. The firing sequence of the token illustrates the execution process of the specification. The firing sequence of the token matches the transition steps of the transition method.

The second example illustrates a DOSL specification of the producer-consumer problem given in Section 2.5.1. There are three object modules which communicate synchronously with each other. We show the semantics of this specification model by two methods. The Petri nets semantics of this problem is given in Figure 3.5. For the transition system method, we focus only on the execution of an object the bounded-buffer. In addition, we assume that the Buffer is initially empty and the producer sends a data item data-1 to the boundedbuffer.

The transition sequences of the object boundedbuffer are:

\[ \sigma_0 = \langle (\text{Item}: \bot), (I: 0), (\text{Buffer}(0): \bot) \rangle \] (before execution)

\[ \langle c_1; c_2; c_3 \rangle, \sigma_0 \rightarrow \langle c_2; c_3 \rangle, \sigma_1 \] (by message accepting statement)

\[ \langle c_2; c_3 \rangle, \sigma_1 \rightarrow \sigma_2 \] (by assignment)

\[ \sigma_2 \rightarrow \sigma_3 \] (by assignment)

where \( c_1 = (\Rightarrow [\text{deposit (Item)}]) \text{ when } \neg (\text{Buffer}(N)) \)

\[ c_2 = \text{I} := \text{I} + 1; \]

\[ c_3 = \text{Buffer(I)} := \text{Item}; \]

and

\[ \sigma_0 = \langle (\text{Item}: \bot), (I: 0), (\text{Buffer}(0): \bot) \rangle \] (initial state)

\[ \sigma_1 = \langle (\text{Item}: \text{data-1}), (I: 0), (\text{Buffer}(0): \bot) \rangle \]
\[ \sigma_2 = <(\text{Item: data-1}), (l:1), (\text{Buffer}(l) : \bot)> \]

\[ \sigma_3 = <(\text{Item: data-1}), (l:1), (\text{Buffer}(l) : \text{data-1})> \]

This specification is also valid since the number of sequences of the state is finite and it does not reach to an abort state.

The meaning of this specification is presented by illustrating the behavior of all three objects, \text{bounded buffer}, \text{producer}, and \text{consumer}. The meaning of the execution of each statement is denoted by the firing of a token in the net, denoted within the parentheses. The abort states are not included in this net.

The Petri net representation of this specification model looks like a symmetric graph. The left half graph represents the behavior (including communication) of \text{the producer} and \text{the bounded buffer}, and the right half represents \text{the consumer} and \text{the bounded buffer}. Place \( s_0 \) is a dummy place which works as a switch for the execution of two methods. In the beginning, we assume that the places \( s_0, s_1 \) and \( s_12 \) have the tokens which represent each object's state. We assume that initially \text{the producer} sends a message \( (s_1 t_1 s_2) \) which, includes a data item, to \text{the bounded buffer} after it produces a data item \( (s_19 t_17 s_1) \). When there is a message in a queue, the method [:\text{deposit}] becomes active \((s_0 t_2 s_3)\). Then \text{the bounded buffer} accepts the message \((s_2 s_3 t_3 s_6)\) and it sends back an acknowledgement to \text{the producer} \((s_6 t_5 s_5)\) so that \text{the producer} continues execution \((s_4 s_5 t_4 s_7)\). Consequently, in \text{the bounded buffer}, a variable \( l \) is increased by one \((s_6 t_5 s_8)\) and the sent data item is assigned to the \text{Buffer} \((s_8 t_7 s_0)\). When the execution of a method [:\text{deposit}] is completed, \text{the bounded buffer} waits for the next message \((s_9 t_8 s_0)\). The behavior of the right half is similar to that of the left half of the graph.
Figure 3.4 The Petri net representation of example
Figure 3.5 The Petri net representation of the producer-consumer problem
3.5 Supplementary Semantics of Message Passing Statements

In addition to the operational semantics of DOSL, the semantics of message passing statements is defined in terms of temporal logic-like formulae. This semantics approach is very close to the algebraic specification techniques which are used to define the semantics of a language. Detailed explanation of such an approach is found in [Ber89].

One of the important features in DOSL is its explicit expressiveness of the communication patterns in the message passing statements. Instead of prefixing the temporal operators to both message sending statements and message accepting statements to specify the communication method, only the message sending statement has a temporal operator as a prefix. The communication pattern of the message accepting statement is automatically determined by the corresponding message sending statement. While two operators □ and ♦ are used for the asynchronous message passing, the operator ○ is used to denote the synchronous message passing method. The communication pattern should be identical throughout the specification model, requiring that □ and ○ cannot appear together within the same model. Such communication constraints are nonfunctional requirements which need to be preserved if the system is implemented. The detailed explanation of the two communication patterns is given in Chapter 2. To supplement the definition of the message patterns, temporal logic-like formulae for each message passing method are defined. In order to define the semantics of the message passing statements, a set of primitive predicates is introduced in Table 3.1.
Table 3.1 Primitive predicates for message passing

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send($O_1,O_2,\text{msg},t$)</td>
<td>an object $O_1$ sends a message $\text{msg}$ to another object $O_2$ at the global time $t$.</td>
</tr>
<tr>
<td>receive($O_1,O_2,\text{msg},t$)</td>
<td>an object $O_1$ receives a message $\text{msg}$ from another object $O_2$ at the global time $t$.</td>
</tr>
<tr>
<td>suspend($O_1$)</td>
<td>an object $O_1$ is in a suspend state.</td>
</tr>
</tbody>
</table>

**Note:** $\text{msg}$ can be replaced by $\text{ack}$ and $\text{reply}$ which stand for an acknowledgement and the requested information, respectively.

3.5.1 The Synchronous Message Passing

In the synchronous message passing, an object which sends a message to a particular object suspends until the partner object sends back a message to it. The receiver object has to send back an acknowledgement to the sender object to ensure that it has received a message although the sender object does not require to receive any information. The object can be active again after it receives an acknowledgement or the requested information. This way of communication may meet a deadlock situation when two objects send messages to each other simultaneously. Moreover, it does not fully support the potential parallelism because an object has to suspend after sending a message until it receives a message [Cor90]. There are two kinds of statements
for the synchronous communication. Although DOSL has two statements for the synchronous message passing, the execution pattern of these two statements is identical because the first statement is assumed to receive the acknowledgement from object $O$ instead of any information to resume the execution. Therefore, the syntax and the meanings of these statements are specified as follows.

• $O$(send $n(e)$ to $O$) --- send a message $n(e)$ to an object $O$.

• $O$(send $n(e)$ to $O$ & get $y$) --- send a message $n(e)$ to an object $O$ and receive the requested information through a variable $y$.

i) send($O_1,O_2,msg,t$) $\rightarrow$ $O$ receive($O_2,O_1,msg,t'$) $\cap$ suspend($O_1$) $\cap$ $t' > t$

Remark) When an object $O_1$ sends a message $msg$ to another object $O_2$, $O_2$ will receive it at time $t'$ and $O_1$ suspends for that time.

ii) receive($O_1,_,msg,t$) $\cap$ receive($O_1,_,msg',t$) $\rightarrow$ $msg = msg'$

Remark) If $O_1$ receives two messages at the same time, these two should be the same message, i.e., $O_1$ cannot accept two different messages simultaneously.

iii) receive($O_1,O_2,msg,t$) $\rightarrow$ $\ast$ (send($O_1,O_2,ack,t'$) $\cup$ send($O_1,O_2,reply,t'$)) $\cap$ $t' > t$

Remark) If $O_1$ receives a message from $O_2$, then $O_1$ will eventually send back an acknowledgement or the requested information to $O_2$.

iv) suspend($O_1$) $\cap$ ((receive($O_1,_,ack,_)$ $\cup$ receive($O_1,_,reply,_$)) $\rightarrow$ $\neg$(suspend($O_1$)))
Remark) If $O_1$ is in a suspended state, it resumes execution after receiving an acknowledgement or reply from other object, i.e., before it receives the acknowledgement, it remains suspended.

3.5.2 The Asynchronous Message Passing

The sender object, in the asynchronous message passing, continues execution instead of waiting until the receiver object receives the message. There are two cases: the sender does not receive any information from the receiver object or the sender object will receive the requested information in the future. Like the synchronous message sending statements, there are two statements in the asynchronous message passing. Since these two statements have different execution patterns, the semantics of each statement is specified separately as follows.

- $\square$ (send $n(e)$ to $O$) \text{::=} \text{send a message without receiving anything}
  
  i) send($O_1$, $O_2$, $msg$, $t$) $\rightarrow$ $\Diamond$ receive($O_2$, $O_1$, $msg$, $t'$) $\land$ $\neg$(suspend($O_1$)) $\cap$ $t' > t$

  Remark) When $O_1$ sends a message to $O_2$ at time $t$, $O_2$ eventually receives a message at the time $t'$, but $O_1$ will be not suspended.

  ii) receive($O_1$, $O_2$, $msg$, $t$) $\cap$ receive($O_1$, $O_2$, $msg'$, $t$) $\rightarrow$ $msg = msg'$

  Remark) Two different messages cannot be accepted at the same time.

- $\Diamond$ (send $n(e)$ to $O$ \& get $z$) \text{::=} \text{send a message to $O$ and receive a message eventually from $O$.}
i) send($O_1, O_2, msg, t$) $\rightarrow$ $\# (receive(O_2, O_1, msg, t') \land \neg(suspend(O_1)))$

Remark) When $O_1$ sends a message to $O_2$ at time $t$, $O_2$ eventually receives the message at time $t'$, and $O_1$ will be not suspended.

ii) receive($O_1, O_2, msg, t$) $\land$ receive($O_1, O_2, msg', t$) $\rightarrow$ $msg = msg'$

Remark) Two different messages cannot be accepted at the same time.

iii) receive($O_1, O_2, msg, t$) $\rightarrow$ $\# (send(O_1, O_2, replay, t')) \land t' = t + n$

Remark) When $O_1$ receives a message $msg$ from $O_2$ at the time $t$, $O_1$ has to send back the requested message within $n$ time units.

Algebraic specification techniques are used as a tool for defining the semantics of part of DOSL. Such semantics are usually expressed as constraints. The semantics defined in this section is provided only for message passing statements as a complement to the operational semantics of DOSL.

3.6 Summary

The meaning of the DOSL specification language is defined using an operational semantics. Two formal methods of language definition, transition systems and Petri nets, are used. To define each category in an abstract syntax of DOSL, we followed Plotkin's transition method which is widely used and is regarded as an effective semantics technique for distributed/concurrent programming languages. The transition rules are defined by showing the execution condition, pre-program parts, post-program parts and state changes. The behavior of a program is interpreted in terms of sequences of states and transitions of states according to program structures. Based
on defined transition rules, the Petri net semantics of DOSL is derived. The firing sequences in the net also represent the operational semantics. These two methods are not independent on each other. The general equivalence between the two methods is proved in [Gol84].

Another approach to semantics is given only for message passing statements. Unlike the other two methods, this method explicitly defines the underlying mechanism which can be interpreted as constraints. Moreover, this approach enables the definition of changes depending on the time. A set of primitive predicates is defined for this static semantics and temporal logic-like formulae for each message passing statement are derived. The underlying constraints of communication methods are specified in terms of such formulae. This semantics of message passing statements is given to supplement two operational semantics approaches.
Chapter 4

Object-Oriented Analysis and Specification

4.1 Introduction

The following methodology presents a framework for using the DOSL specification language. It includes a method that helps the user write a specification in DOSL.

There are two distinct approaches to software development: functional-oriented and object-oriented. While the desired system is decomposed into a set of interacting functional units in the functional-oriented approach, the system is decomposed into a set of objects and their operations in the object-oriented approach. The object-oriented approach has received high interest and is regarded as a good approach for both sequential systems and distributed systems development. Supporting methods and tools for functional-oriented approaches are numerous and are widely used; however, object-oriented methods and tools are lacking in part because interest in the object-oriented approach is relatively new.

4.1.1 Object-Oriented Development

Object-oriented approaches to software development have received increased emphasis since the early 1980s. These new software development methods are expected to be used widely due to features such as information hiding, modularity, abstraction, and localization [Boo87]. Initially, object-oriented methods were applied primarily during the implementation phase using object-oriented languages. Recently, object-oriented paradigms have been applied to earlier phases of the software development process. Numerous efforts regrading object-oriented design approaches are found in [Bai89], [Boo86], [Boo87] and [Mey88]. More recently, object-oriented
analysis techniques are being used to initiate the object-oriented software development process [Coa90].

The real world problem is bounded by the identification of the objects, their properties, their actions, and the relationships among the objects in object-oriented systems. Thus, the resulting format is similar to that of an abstract data type because an object encapsulates its data and actions. The structuring of a system around the real-world objects supports the desirable traits of abstraction and information hiding. It provides a stable foundation for software development and enhances the maintainability of the system due to the localization of the objects properties and actions. A major benefit of this approach is that it allows the evolution of a system in terms that are understandable to the user. Another advantage is that the implementation details of an object can be changed without impacting the rest of a system, thereby increasing maintainability. Inheritance is a powerful feature that provides for the reusability and extendability of software components [Mey88]. The main difficulty of this approach is the identification of appropriate objects and their operations from the initial user requirements. A requirements analysis phase is needed to precede the explosion of the design from an object-oriented viewpoint.

Systems designed from this approach tend to be flat instead of hierarchical. Each module denotes an object or class of objects. Object-oriented development builds on the concepts of abstract data types. An operation on an object may be classified as a constructor which alters the state of an object, a selector which evaluates the state of an object, or an iterator which permits part of the object to be visited. Each object may be viewed externally by other objects from its specification or internally from its implementation details. The actual object-oriented development process consists of identification of objects and their attributes, identification of operations performed on
or by the objects, identification of the visibility of the objects in relation to the other objects, solidification of the boundary between the inside and outside view by establishing the interface, and finally the implementation of the objects.

Methods and techniques for object-oriented system development are insufficient. The general lack of methods includes methods which support the entire life-cycle and methods which are useful for large scale system development.

4.1.2 Existing Methodologies for Object-Oriented Systems

Requirements analysis typically begins with a narrative document. Use of a narrative requirement document for the information domain provides many difficulties for object-oriented analysis because of the difficulty of identification of appropriate objects and their actions. To help address such problems, many object-oriented analysis methods are initiated by domain information that is input in a structured format. In [Lad88], several methods to assist with the selection of objects are described. One method uses data/control flow diagrams and combines the process bubbles, stores and flows into objects. A second technique identifies the entities in entity-relationship (ER) diagrams as the objects. A third method is the concurrent use of data/control flow diagrams and state transition diagrams with entity-relationship models to identify the objects. Many methods and techniques are developed to derive an object-oriented specification model. Limited automated support exists for some of these methods; however, in general they require a manual derivation process. Providing formalism is a primary problem in most methods for object-oriented system development. Among object-oriented methods, four methods are discussed below.

The first object-oriented technique is the Jackson Structured Development (JSD) developed by M.A. Jackson [Jac83]. The JSD methodology is presented as a full life
cycle methodology which begins with the requirements analysis phase and continues through the implementation phase, but it is not pure a object-oriented methodology because it does not support inheritance and message passing. JSD consists of seven steps: entity action step, entity structure step, initial model step, function step, system timing step, and implementation step. The identification of objects is done from the first two phases. JSD has been suggested as an applicable front end methodology to the other object-oriented methodologies.

Bailin [Bai89] has introduced a method to derive an object-oriented specification composed of a set of entity data flow diagrams (EDFD). The EDFD is similar to the DFD except nodes are represented by active entities or functions and arrows between each node are denoted by passive entities. An entity and its operations are extracted from the process’s function name action-object. The extracted entities are divided into two groups: active and passive. An entity which works as an actor becomes an active entity and the remaining entities become passive entities. The entity-relationship (ER) model [Che76], which contains all entities of a system and illustrates the relationships between each entity, is used as another source. Every entity in the ER model must appear as an active or a passive entity. The advantage of this method is that the produced EDFD is easy to understand and explains the structure of the entities. A disadvantage of this approach is its lack of automatic supports.

Booch, one of the pioneers in object-oriented development concepts, showed a simple method to construct an object-oriented specification from data flow diagram [Boo86]. A set of entities is extracted from sources, sinks and data stores. Only external entities are taken for solution objects, but the real active objects of the system are missed. This method is useful if only a simple high-level data flow diagram is given as a source.
In [War89], the concept of conversion from extended data flow diagrams which is designed for real-time systems to an object-oriented model using model-building heuristics is given. The objects are instances of abstract data types extracted from the data flow diagrams by collecting low level functions and data stores into high level transformations. The ER model is also used as a tool for identification of the objects.

Cordes [Cor88a, Cor88b] introduced a methodology which derives an object-oriented specification model from natural language documents using parsing techniques. It automatically extracts objects, operations, and other information from an initial document. The traceability possible in the methodology helps to improve the quality of the derived specification. The main problems with this methodology result from the inherent ambiguity in the user requirements documents, i.e., the correctness of the produced model is totally dependent on an initial document which is usually incomplete and ambiguous. The final model which looks similar to Booch's representation [Boo87], is used as an architectural model.

In this chapter, we present a technique that provides knowledge-based assistance to object-oriented analysis. It is designed to assist the specifier with the derivation of objects, actions and visibilities. Two different levels of data flow diagrams, a functional-oriented approach, are used to obtain the domain information. The technique also supports the derivation of an information model in the form of an entity-relationship diagram. In addition, it provides an automated first pass to an object-based architectural design. The overall feature of the technique is illustrated in Figure 4.1. Each step in the diagram is discussed in the following sections. The input of this method is introduced in Section 4.2. In Section 4.3, we describe the technique to represent the domain information and procedures for creating the knowledge base
environment. The derivation of the ER model and user document are described in Section 4.4 and 4.5, respectively. The building of an object-oriented specification model is described in Section 4.6. Finally, Section 4.7 contains a summary.

---

**Figure 4.1** The overview of the object-oriented analysis technique

4.2 Input Model

For the first step of software development, it is important to identify a model of the desired system which can be used as a blueprint through the entire software development. This model should be readily understood by the user and the system analyst. One of the most popular notations used to model systems is the data flow diagram,
designed by De Marco [DeM78]. Numerous varieties of data flow diagrams have also been developed [You89]. The data flow diagram, a graphical representation, emphasizes the stream of data. It decomposes a system according to functions and emphasizes the data transformation without considering the sequence and control aspects.

The initial step of construction of a data flow diagram is to establish the context diagram (level 0) which is the highest level of abstraction; thus, it describes an overview of the system's function [Kow88]. This top level, the context diagram, consists of sources, sinks, one main process, and input/output data flows. The level 01 data flow diagram is derived by decomposing the context diagram's process into different processes and data stores without altering sources and sinks. The next level data flow diagram is formed by decomposing its high level diagram. This decomposition process continues until each process is at a primitive stage. Different levels of data flow diagrams are used to express a system at different levels of abstraction. The data flow diagram consists of five components: source, sink, process, data flow, and data store. Each component is explained below.

Components of a data flow diagram

* Source, Sink

a square is used to represent the source and sink of the data. When the context diagram is devised, these two components are built around the one process. Sources and sinks are not added or deleted during the further decomposition.

* Process

a circle is the symbol used to represent a process. One process is denoted by one circle. If there are a group of processes which performs the same job at the same time, multiple circles are used to represent the group of process.
• **Data store**

Two parallel lines are used to represent the data store. The data store is where data is stored and retrieved by the processes. In an actual system, a data base is a typical example of a data store.

• **Data flow**

An arrow is used to represent the data flow. The data flow acts as a pipeline where the data is transferred between each component. There are two kinds of data flows: elementary elementary and group element. An elementary element has only one type of data, and a group element is a combination of some similar elements or other group data elements. Data flows can be either discrete or continuous. Continuous data flow is illustrated using a double-headed arrow.

• **Data dictionary**

Although a data dictionary is not a primary element in the data flow diagram, it represents the contents of each component of the data flow diagram, i.e., a data flow diagram needs a data dictionary to define the exact contents of data flows, data stores, and commonly used processes. A quasi-formal grammar notation is used to describe the information [Pre87].

In [Cut88], data flow diagrams are classified into two groups: physical data flow diagram and logical data flow diagram. The physical data flow diagram explains "how the system operates" and the logical data flow diagram denotes "what the system accomplishes". The logical data flow diagram's process has only a function name because it shows the function of the system without considering the actor of that function [Cut88]. If only a set of logical data flow diagrams exists, it is...
impossible to extract active objects (actors or agents), since an actor of the function is not specified in the data flow diagram. So, the final object model will have only passive object (server) modules. Unlike the logical data flow diagram, the physical data flow diagram's process has both a process name and a function name, where a process name denotes the actor of the function in the process. To identify a meaningful actor object model, we need at least a level 01 physical data flow diagram so that the lower level logical data flow diagram's functions are related to a level 01 data flow diagram's processes. A concurrent system requires the parallel processing (concurrency) model instead of conventional sequential processing model. To develop a concurrent object model, the active object modules are necessary to identify the control or supervisor modules of the system [Boo87], [Gom90].

The problem description of a hypothetical system, a student registration system, is given below and different levels of data flow diagrams for this problem are given in Figures 4.2.1 and 4.2.2.

A college plans to develop a telephone registration system. Initially, each department in the school sets up the opening courses for the coming semester and submits the list of courses to a course scheduler who adjusts the overall courses such as time and class rooms. When the course schedule is ready, a package is printed and sold at the book store to the students. The student who wants to preregister needs to use the telephone to add the courses. When the phone receiver gets the student's request, he adds the student's requested courses in the student file and the course file. During the registration period, the student pays the fee to the registration clerk who validates the student id before he/she accepts the payment. The registered student names are stored in a student file and the list of registered students for particular courses is reported to the department.
Level 01 data flow diagram

Figure 4.2.1 The high levels of data flow diagrams
Figure 4.2.2 The low level of data flow diagram
4.3 Building of the Knowledge Base Environment

The first step is to build a knowledge base environment in which the domain information is stored and new facts are inferred according to heuristic rules. The initial information in the knowledge base consists of an internal representation of the data flow diagrams. Based on this internal information, additional facts are derived to provide the information for the object model.

4.3.1 Internal Representation of the Data Flow Diagram

In this technique two levels of data flow diagrams, the level 01 data flow diagram and the lowest available level data flow diagram, are used as input. The higher level data flow diagram shows the object abstraction and the lower level data flow diagram contains information about the functions of the system. Both levels are needed in this technique. Conversion of the data flow diagram components into a knowledge base representation form is the initial step of the technique. The internal form of the data flow diagram is adopted from [Cor88a] which describes salient requirements of knowledge base representation based on the following criteria. First, the representation should be generic so that it can be applied to any kind of application. Second, it should be able to compensate for missing or incomplete information of the source because the source information often lacks a portion of the information required for specification development. The third requirement is that the representation should be easily modified.

The data flow diagram's five components -- source, sink, process, data store, and data flow -- are converted into knowledge base facts. The general formats of the data flow diagram's internal representation are given in Table 4.1.
Table 4.1 The level 01 data flow diagram’s internal representation

| source1(id,[source_name]). |
| sink1(id,[sink_name]). |
| process1(id,[process_name],[function_name]*). |
| dst1(id,[datastore_name]). |
| dfwl(id,[dataflow_name]). |

Note: * [function_name] := [action, object]

Each level of the data flow diagram has a distinguishing suffix number: 1 indicates the high level data flow diagram, 2 indicates the lower level data flow diagram. The format source1(id,[source_name]), for instance, indicates a level 01 data flow diagram’s source. This format provides easy element identification, facilitates building of the knowledge base, and allows easy modification of information. If some component of a data flow diagram does not have the required information, it is null in the internal representation. The internal form of each level of the example data flow diagrams is illustrated in Table 4.2.

4.3.2 Object Identification

The identification of a set of objects is the next step. An object is an entity which exists in the real world and has a state whose behavior is explained by the operations (actions) that it performs or is performed by [Boo86]. An entity is an object that exists and is distinguishable from another object [Che76]. Intuitively, an object is more logical than an entity. For example, we may say that sorted_file is an object, but it may not be an entity. But, many researchers agree that an entity is an object, and an object is an entity. In this work, we use these two terms interchangeably.
Table 4.2 The internal form of the data flow diagrams

source1(f_01,[student]).  sink(t_01,[department]).

dst1(s_01,[student_data]).  dst1(s_02,[course_data]).  dst1(s_03,[book_store]).

process1(p_10,[phone_accept_section],[receive,student_request]).
process1(p_20,[registration_section],[perform,registration]).
process1(p_30,[schedule_section],[plan,schedule]).

dfw1(d_01,f_01,p_10,[]).  dfw1(d_02,p_10,s_01,[]).

% LEVEL 02 DFD
source1(f_01,[student]).  sink(t_01,[department]).

dst2(s_01,[student_data]).  dst2(s_02,[course_data]).
dst2(s_03,[department_data]).  dst2(s_04,[book_store]).
dst2(s_05,[cash_box]).

process2(p_11,[phone_receiver],[receive,receiver]).
process2(p_21,[phone_receiver],[update,receiver]).
process2(p_22,[registration_clerk],[validate,student_id]).
process2(p_23,[registration_clerk],[accept,payment]).
process2(p_31,[scheduler],[maintain,department_data]).
process2(p_32,[scheduler],[plan,course_schedule]).
process2(p_33,[report_clerk],[print,course_schedule]).
process2(p_34,[scheduler],[merge,data]).
process2(p_35,[report_clerk],[print,reports]).

dfw2(d_01,f_01,p_11,[student_id]).  dfw2(d_02,p_11,p_12,[request]).
dfw2(d_03,p_12,s_01,[student_record]).  dfw2(d_04,f_01,p_21,[package]).
dfw2(d_05,p_21,p_22,[package]).  dfw2(d_06,p_22,p_23,[package]).
dfw2(d_07,p_22,s_05,[money]).  dfw2(d_08,p_23,s_01,[student_record]).
dfw2(d_09,s_01,p_34,[student_list]).  dfw2(d_10,p_34,p_35,[merged_data]).
dfw2(d_11,p_35,f_01,[receipt_and_id]).  dfw2(d_12,p_35,s_01,[student_list]).
dfw2(d_13,s_01,p_31,[department_data]).  dfw2(d_14,p_31,s_03,[department_data]).
dfw2(d_15,s_03,p_32,[department_data]).  dfw2(d_16,p_32,s_02,[course_schedule]).
dfw2(d_17,s_02,p_34,[course_schedule]).  dfw2(d_18,s_02,p_33,[course_schedule]).
dfw2(d_19,s_02,p_12,[course_schedule]).  dfw2(d_20,p_33,s_04,[book_store]).
dfw2(d_21,s_04,f_01,[course_schedule]).
Booch classifies an object as actor, agent, or server, based on the relationships with the other objects. If an object operates on other objects without receiving any actions, it is an actor object. The server object merely receives actions by actor objects. If an object performs actions or can have actions performed on it by other objects, it is classified as an agent object [Boo87]. We can extract objects from any component of the data flow diagram. A process name, source, sink, and data store name are potential sources of objects. Also the function name action-object contains an object. These objects are the problem domain objects in this technique. The selected objects from the level 02 data flow diagram are listed in Table 4.3.

<table>
<thead>
<tr>
<th>book_store</th>
<th>cash_box</th>
<th>course_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>course_schedule</td>
<td>data</td>
<td>department</td>
</tr>
<tr>
<td>department_data</td>
<td>payment</td>
<td>phone_receiver</td>
</tr>
<tr>
<td>registration_clerk</td>
<td>report_clerk</td>
<td>reports</td>
</tr>
<tr>
<td>request</td>
<td>scheduler</td>
<td>student</td>
</tr>
<tr>
<td>student_data</td>
<td>student_id</td>
<td></td>
</tr>
</tbody>
</table>

4.3.3 Extracting Actions

The action represents the behavior of the object. The origin of the action from the data flow diagram is the process name that consists of an action-object pair. We identify a subject of the action from the process name, and an object (grammatical meaning) of the action from the function name action-object pair. We also extract implied actions from the relationship of each component of the data flow diagram. Since the data flow diagram's process only contains the main function, it is possible to lose some actions from the initial requirements during the design of the data flow
diagram. For example, the original user requirements sentence, "The clerk saves a file into the data base." may not be identified as one function of the data flow diagram because the data flow between a process and a data store implies that action. So, we extract this action from the data flow connections if the action does not overlap with the function name of the process. To define correct corresponding actions of the object, we require a well-defined data flow diagram whose processes contain at least one function name action-object. Some level 01 data flow diagrams may not contain the process function name, because they only represent the objects of the system. In such a case we use the lower level data flow diagram to extract the actions of high level data flow diagram's objects. Figure 4.3 shows the related actions of an object.

```
Object :: [registration_clerk]

Action → save [payment] [student_id]
Action → validate [student_id]
Action → accept [payment]
Action → register [student_id]
```

Figure 4.3 Actions

4.4 Construction of an ER Model

The ER model, developed by Chen [Che76], is widely used for the conceptual model of a system. It has many benefits for explaining a set of system entities and relationships, using a graphical representation. In this methodology, we derive an ER model for the user benefit. From the extracted entities, including the names of
sources and sinks, and the structure information of the level 01 data flow diagram, we construct an ER model. The entities which are connected by data flow have a relationship to each other. In addition, from each process we can extract two entities (actor and server) and their relationship. The multiplicity of the relationship is not considered.

The ER model can be clustered into high level diagrams. Clustering makes an ER model an abstraction so that the end user and the system developer can understand the system at the top level view [Teo89]. A more detailed ER model can be derived with level 02 data flow diagram's entities, but, in this work, we show only the most abstract view of the system to the user. We extract the relationships from the knowledge base. Each data flow between two components implies the relationship of these two entities. The derived ER model from the level 01 data flow diagram is showed in Figure 4.4. This ER model provides an additional view of the problem domain.

![Figure 4.4 The entity-relationship diagram](image-url)
4.5 Regeneration of User Document

Traceability is an important factor which we need to consider during the requirements specification phase. As the first phase of software development, the requirements specification tends to be changed frequently according to the user requirements' changes.

We support the traceability by regenerating a document from the DFD's internal representation information and the extracted ER model's information. The document describes the system with simple English sentences which are concise, consisting of a subject, a verb, an object, and a prepositional phrase. Each process can derive one sentence directly without any interpretation. For instance, a fact process2(p_11,[phone_receiver], [receive,request]) can generate a straightforward sentence "phone receiver receive request." The other sentences are generated by adding verb phrases into the relationship as introduced from the ER model relationship step.

Using the regenerated document, we provide an opportunity for the user to check the requirements. If the user finds ambiguous information, the data flow diagram can be modified accordingly. Thus, the regenerated system document is a useful mechanism to help to verify the correctness and the information in the data flow diagrams. It can also be used as a source for other methodologies [Cor88a] which derive an object-oriented specification model from a user document.

4.6 Construction of An Object Specification Model

Once the knowledge base has been established, three steps are required in order to construct an object specification model. The first step is the classification of objects as active or passive objects. The next step is the derivation of the relevant
information for each object model. The final step is the actual generation of the object model. Each step is described in the following sections.

4.6.1 Classification of Objects

From the set of objects which is derived from the data flow diagram, we define two groups, problem space objects and solution space objects, according to the characteristics of each object. There does not exist a single definition of problem space objects and solution space objects. Instinctively, a solution space object is an entity which is related to an event of the system directly, and the remaining objects are problem space objects. All objects that perform as actors, agents, or servers within the system are solution space objects. We define rules to select the solution space objects from the objects. In general, most entities which are extracted from the data flow diagrams are solution space objects because the other entities have been eliminated during the design of the data flow diagram. So, except for the name of the source and sink, the remaining objects typically become solution space objects.

From the solution space objects, we define two groups of objects: active and passive objects. An actor object or agent object becomes an active object and a server object is defined as a passive object. In general, an active object represents the person, hardware object, place, or controller of the system which appears as a process name in the data flow diagrams, i.e., the name of process becomes an active object. The objects which come from the data flow diagram information, with the exception of the process name, become the passive objects which merely receive operation(s) from the other objects. The algorithm for classification of objects is given in Figure 4.5.
Identify all Objects from the DFDs

If the object is from the source or sink
   classify as a problem space object
else
If all operations in the object are of a passive type
   classify as a passive object
else
   classify as an active object

Figure 4.5 Algorithm for the object identification

4.6.2 Definition of the Object Model

In this step, the knowledge base environment for each object is developed in order to generate the object model. From the existing knowledge base information, we assert additional facts which are required in order to construct each object module. A type fact defines the type used for the development of the associated object. An inherit fact identifies the class objects of the current object. The object and action pairs are identified with an obj_act fact. A vi_object fact defines the other objects in the system that are related to a given object. By indicating the related objects, the visibility of each object can be established. We define rules to infer new facts from given information. The procedure to define each fact is described below.
**Type Definition**

The type of an object is determined by the characteristics of the object. We divide the objects into two types: active or passive objects as described in Section 4.6.1. An active object (an actor or agent) defines a separate entity within the system that is capable of initiating independent actions. A passive object (a server) defines a specific data representation that is manipulated by the other objects. It only receives the actions that are instigated by the active objects. Thus, the rule to define type is that objects which come from the process names are active objects and the other objects are passive objects.

**Inheritance Identification**

One of the most powerful features of object-oriented paradigms is inheritance which provides the reusability and extensibility of software components [Mey88]. By classifying the objects according to their properties and actions, subclass objects can inherit the actions from their class object. Using the inheritance property, a new object module can be built without defining all of its actions if it has an existing class object module.

The data flow diagrams do not specify the hierarchical relationship of the objects explicitly. Identification of class and instance objects from the data flow diagram information is difficult, particularly when a logical data flow diagram which does not have a process name explicitly is used. But if there is a set of physical data flow diagrams, we classify such relationships. A heuristic rule is necessary to identify the hierarchical relationship of the objects. A class object and instance objects can be derived from the relationship of the level 01 data flow diagram and lower level data flow diagram. In the example data flow diagrams,
has been extended into process2(p_31,[scheduler],[]), and process2(p_35,[report_clerk],[]), then scheduling_section is a class object of scheduler and report_clerk. Inheritance relationships exist between a class object and its instance objects. A fact class is asserted to specify the class object of a given object.

Collection of Actions

Actions of an object are primary obtained from the function name action-object as described in Section 4.3. A primary difference between the data flow diagram decomposition method and the object-oriented method is the principle of aggregation. The data flow diagram groups the functions together according to their characteristics. The object-oriented method groups functions according to the object on which they perform or by which they are performed [Bai89]. We gather the related actions of each object. For this step, additional rules are defined. The verb appearing in the function name becomes the action of two objects. One is the actor of that action and the other is the receiver of that action. While a process name becomes the actor object, the object from the function name becomes the receiver object. The meaning of this action is different according to the type of an object. The active object uses its actions to activate the corresponding passive objects. The passive object identifies its behaviors with these actions. For the implementation, such actions become functions or procedures. A new fact, obj_act is created to represent an object and its actions pairs in the knowledge base.

Visibility of the Object

One of characteristics of object-oriented development techniques is the representation of the object visibility. Visibility of an object can be expressed by
specifying the objects which interact with a specific object. The visible fact indicates the other objects which are necessary to explain the action of a given object. The visible objects in an active object are a set of passive objects which receives the actions from that active object and a set of active objects which is connected with a given object by a data flow in the data flow diagram. These related passive objects are activated when they receive a message from an active object. The passive object specifies its related active objects as visible objects; however, this passive object cannot see its visible objects but can be seen from them. The visibility between passive objects cannot be extracted from the data flow diagram information.

4.6.3 Generation of the Object Model

The generation of the object model is the final step. The environment for the generation of this model is the knowledge base which contains all necessary facts derived from the above steps. The developed model has the form of a definition part. It is possible to construct the object model without regard to the type of the object. Each object module is treated as an abstract data type which encapsulates data and operations so that only internal operations can manipulate the defined data. For the sequential processing system, the set of passive type objects provides a useful initial object model. For a real-time system or a concurrent system, both the active and passive object modules are required to identify the concurrent processing explicitly. The active object becomes a monitor module which controls the executions of related passive object modules. The actions of an active object module are regarded as triggers of the passive object module, i.e. the passive object module can be in an active state after receiving the message from its monitor module. The sequential processing system does not need the monitor module, since the flow of execution follows the sequential order of the coded modules.
There are many similarities between object-oriented systems and concurrent programming systems. While an object-oriented system consists of object modules which communicate with each other by message passing, a concurrent system consists of a set of processes which executes in parallel with inter-process communication. To expand the application of this technique to concurrent systems, determination of the active objects is a very important step. In our approach, we allow the user to determine whether both active and passive object models are generated.

Finally, we specify the body of the defined object model in a program design language. We use a format that is compatible with DOSL. The general form of an object module is illustrated in Figure 4.6 and the complete object specification model from the example data flow diagrams is given in Section 4.6.4.

ObjectModule : \( object\_name \)  
\( type \quad \rightarrow \quad active \ or \ passive \)  
\( class \quad \rightarrow \quad parent \ object \)  
\( visible \quad \rightarrow \quad visible \ objects \)  
\( method \quad \rightarrow \quad action \)  
\( method \quad \rightarrow \quad action \)  
......  
End.

Figure 4.6 The general form of the object module
4.6.4 Specification Model

The generated specification of the student registration system is given in the following sections.

4.6.4.1 The Active Object Modules

ObjectModule : [phone_receiver]
  type --> active
  class --> [phone_accept_section]
  visible --> [request]
  method --> save [request]
  method --> get [request]
  method --> receive [request]
  method --> update [request]
End

ObjectModule : [registration_clerk]
  type --> active
  class --> [registration_section]
  visible --> [payment] [student_id]
  method --> save [payment] [student_id]
  method --> validate [student_id]
  method --> accept [payment]
  method --> register [student_id]
End

ObjectModule : [report_clerk]
  type --> active
  class --> [schedule_section]
  visible --> [course_schedule] [reports]
  method --> save [course_schedule]
  method --> get [course_schedule]
  method --> send [receipt_and_id] [student_list]*
  method --> print [course_schedule] [reports]
End

ObjectModule : [scheduler]
  type --> active
  class --> [schedule_section]
visible --> [scheduler] [department_data] [course_schedule] [data]
method --> send [merged_data]*
method --> save [department_data] [course_schedule]
method --> get [data] [course_schedule]
method --> maintain [department_data]
method --> plan [course_schedule]
method --> merge [data]

End

4.6.4.2 The Passive Object Modules

ObjectModule : [course_schedule]

type --> passive
class -->
visible --> [scheduler] [report_clerk]
method --> is plan by [scheduler]
method --> is print by [report_clerk]
method --> is save by [scheduler] [report_clerk]
method --> is get by [report_clerk] [scheduler]

End

ObjectModule : [data]

type --> passive
class -->
visible --> [scheduler]
method --> is merge by [scheduler]
method --> is get by [scheduler]

End

ObjectModule : [department_data]

type --> passive
class -->
visible --> [scheduler]
method --> is maintain by [scheduler]
method --> is save by [scheduler]

End
ObjectModule : [payment]

type --> passive
class -->
visible --> [registration_clerk]
method --> is accept by [registration_clerk]
method --> is save by [registration_clerk]

End

ObjectModule : [reports]

type --> passive
class -->
visible --> [report_clerk]
methods --> is print by [report_clerk]

End

ObjectModule : [request]

type --> passive
class -->
visible --> [phone_receiver]
method --> is receive by [phone_receiver]
method --> is update by [phone_receiver]
method --> is save by [phone_receiver]
method --> is get by [phone_receiver]

End

ObjectModule : [student_id]

type --> passive
class -->
visible --> [registration_clerk]
method --> is validate by [registration_clerk]
method --> is provide by [student]
method --> is register by [registration_clerk]
method --> is save by [registration_clerk]

End

Note : [data]* comes from the data flow in the data flow diagrams. We do not regard it as an object.
4.7 Summary

This chapter describes a technique which provides assistance for the derivation of an object model from a set of data flow diagrams. The overall summary of this technique is given as an algorithm in Figure 4.7. This method extracts objects and actions from the data flow diagram, constructs an ER model, and builds an object model using a knowledge base environment.

begin
  execute Identify the active objects and passive objects
  for each object defined do
    execute Collect information for each object with
    Object = object
  od
  for each object defined do
    execute Generate an object model for active type objects
    execute Generate an object model for passive type objects
  od
end

Figure 4.7 An algorithm for the object model construction

With this technique, the user can derive the object model automatically from two different levels of data flow diagrams. Most object-oriented analysis methodologies have little automated support. The benefits of the technique presented in this methodology can be divided into three aspects.

First, the use of a knowledge base system provides many advantages for system development. The data flow diagram's graphical notations are represented in internal representation form without losing information. New information can be extracted from existing information according to the defined heuristic rules.
A second benefit of this technique is its derivation of an ER model, which contains the entities and the relationships among the entities. This constructed ER model is a useful tool for the database design.

A third benefit is that the syntactic format of the produced object model maps to DOSL. Since the information of the data flow diagrams is not sufficient to describe the internal behavior of objects, the produced model represents the definition part of the model.

A limitation of this method is that the input data flow diagrams must be nonambiguous. The information in the data flow diagram is often not sufficient. Some data flow diagrams do not have a process name or a data flow name. In this case, we only use the existing information to derive the object model, which may be an ill-designed one. The data flow diagram which has a name and a function name together in each process is preferred in our system. Another current limitation is that each process in the data flow diagrams must contain only one function name.
Chapter 5

An Integrated Modeling Methodology for Distributed Systems

5.1 Introduction

Distributed computing systems are systems in which multiple processors with their own memories run independently by communicating with each other. The design of distributed systems is difficult to achieve as the execution patterns of distributed systems are typically more complex than those of non-distributed computing systems. Thus, research toward the development of design methodologies for distributed systems is needed. One of promising approaches is applying object-oriented techniques to the design of distributed systems so that the power of computation and modeling increase simultaneously.

On the other hand, the need for effective techniques to design large systems increases, as complex requirements cause system size to become large. Formal methods which span the analysis and the design phases are needed for large scale systems. There is currently more research toward the development of notations and techniques for specification models that to the development of support tools for the large-scale specification [Som89].

In this chapter, we present a specification methodology from a distributed object-oriented viewpoint. It integrates information from multiple models to specify objects, object behavior and relationships among objects. Multiple modeling techniques are typically used to specify a system as different models specify the system from different viewpoints. When a system is specified by a set of different models, correct integration of such information in order to derive a system specification is a critical task. This methodology is an extension of the technique introduced in Chapter 4. A goal of
the methodology is to provide assistance to the process of specifying a formal object-oriented specification from graphical representation specification inputs, including data flow diagrams, state transition diagrams and Petri nets. Input models of this methodology are introduced in Section 5.2. The methodology is discussed in Section 5.3. Finally, Section 5.4 contains a summary.

5.2 Input Models

Since the desired system is typically not represented by a single model, multiple representative models are used in order to specify a system from different viewpoints. For instance, the initial problem is frequently represented with informal representation techniques, such as data flow diagrams and entity relationship diagrams. As the definition of the requirements proceeds, more formal methods, such as state transition diagrams and the Petri nets, are used to show control and behavior. Among many different modeling techniques, we have selected three widely used models, data flow diagram, state transition diagram and Petri nets, to specify the initial problem domain information. However, as these various techniques represent different viewpoints of the application, a methodology which combines the requirements from the different models to produce an integrated specification is needed. From the numerous versions of each model, we have selected a representative format. Data flow diagrams were introduced in Chapter 4. The other two models used as input formats for this methodology are introduced briefly.

Petri Nets: Petri nets, designed by C.A. Petri [Pet62], have been widely used as tools for the design of communication protocols [Son88] and distributed/concurrent computing systems [Pet81]. The power of modeling a system with Petri nets has been increased by extensions to the original Petri net model. Extended formats of Petri nets
include Colored Petri nets [Jen81], Predicate/Transition nets [Gen78] and Time(d) Petri nets. The advantages of Petri nets include powerful modeling ability, ease of understanding and validation, support of theoretical techniques and the possibility of automation [Son88]. In addition, Petri nets provide a powerful formal mechanism for representing the specification. As Petri nets contain the rules that control the dynamic changes to object attributes and relations, they are important to this methodology.

Transitions provide a model of actions and places represent the conditions associated with the actions. The occurrence of an action is shown by firing a transition. The model of a sequence of actions is shown by a sequence of transitions [Pet81]. Petri nets are especially suited to the specification of distributed systems due to their ability to specify concurrency requirements. They exhibit nondeterminism in that when more than one transition is able to fire, any one of them may fire. The executability of Petri nets has resulted in their use as an integral part of numerous prototyping systems [Kra87].

A marked Petri net, $C$, [Pet81] is formally defined as

$$C = P, T, I, O, M$$

where

\[
P = p_1, p_2, \ldots, p_n, \text{ a finite set of places}
\]

\[
T = t_1, t_2, \ldots, t_n, \text{ a finite set of transitions}
\]

\[
I : T \rightarrow P, \text{ a mapping from transitions to bags of places (input function)}
\]

\[
O : T \rightarrow P, \text{ a mapping from transitions to bags of places (output function)}
\]

\[
M : P \rightarrow \{0,1,2,3,\ldots\}, \text{ the set of token in the places.}
\]

There are two approaches to the use of Petri nets in software development. One approach is to view the Petri net model as an analysis tool where the system properties
are analyzed and modeled in Petri net form. Petri nets are then analyzed for such properties as safeness, boundness, liveness, and reachability. A second use of Petri nets in specification and design is to use them for the entire specification and design process, thus requiring the transformation of Petri net representations into systems. This second approach is taken in the specification language, SEGRAS [Kra87], which combines the use of abstract data types and Petri nets in a common syntactic and semantic framework.

State Transition Diagram: The state transition diagrams are used for specifying and designing computer systems due to their power to show response to a given stimulus. They are chosen as one of the input models because they are fundamental to many existing specification methods, including REVS [Alf85] and RLP. They do not provide for specification of parallelism as is possible with Petri nets, but they are useful for the specification of the behavior of objects that respond to only one stimulus at a time. For real-time systems, DARTS [Gom90] uses the data flow diagrams and the state transition diagrams as input. While DARTS is not based on the object-oriented approach, it supports the concurrent execution of a system. Benefits of combining state transition diagrams and object models are discussed in [Tys90].

A state transition diagram defines a finite state machine (FSM) which is formally defined as

\[ FSM = \{ J, I, T, S, F \} \]

- \( J \) is a finite nonempty set of states
- \( I \) is a finite nonempty set of inputs
- \( T \) is a transition function from \( F \times I \rightarrow I \)
- \( S \) is the initial state, \( F \) is the set of final states
5.3 Object-Driven Specification Methodology

This object-driven methodology is an integrated, formalized method for identification of objects, object properties and object behaviors from multi-model formats. It addresses the extraction of objects, actions and relationships from the problem domain with emphasis on the specification of the characteristics of distributed systems. The object identification methodology is supported by a knowledge base that provides for automated analysis and reasoning about objects and their relationships. The final object model is represented in DOSL which provides a formal mechanism for representing the object information. It also provides constructs that allow for refinement of the specification.

The methodology consists of the following five steps:

1. Develop the graphical representations.
2. Convert each representation into an internal form.
3. Build a knowledge base.
4. Synthesize the input information.
5. Generate an object-oriented specification.

Each step is discussed in the following sections.

5.3.1 Develop the Graphical Representations

We explain the procedures of this methodology by applying it to an example, an elevator system. We have selected this problem as it is widely used as a specification example [Kam87], [Ghe91]. In Table 5.1, the initial user requirements of the elevator system are described. A portion of the representation of this problem with the data flow diagram, state transition diagram and Petri net is illustrated in Figures 5.1.1, 5.1.2.
and 5.1.3, respectively. While the data flow diagram emphasizes the overall view of a system, the Petri net and the state transition diagram describe a single component of the system, the elevator movement. The dynamic behavior of process 3.6 is specified by a state transition diagram and a Petri net. The specification of the other parts of this elevator system with the Petri nets and the state transition diagrams is not included. The methodology is defined in Sections 5.3.2 through 5.3.5.

Table 5.1 Requirements for an elevator system problem

<table>
<thead>
<tr>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>An elevator system with $n$ elevators is to be installed in a new building which has $m$ floors. To make it an automatic system, a software system is required. The constraints of this elevator system are as follows:</td>
</tr>
<tr>
<td>1. There are two kinds of buttons, internal buttons and floor buttons. The passengers make their requests by pressing buttons. An elevator has a set of internal buttons which indicate floors. They are set when pressed and reset when the elevator reaches the corresponding floors.</td>
</tr>
<tr>
<td>2. Except for the first and the top floors, there are two floor buttons on each floor, one for upward and one for down elevator movement. These buttons are set when pressed by passengers.</td>
</tr>
<tr>
<td>3. If there is no request, the elevator remains at its final floor and keeps waiting until the next request.</td>
</tr>
<tr>
<td>4. Each request is serviced with equal opportunity, and there should not be a starvation situation.</td>
</tr>
<tr>
<td>5. The system will service all requests within elevators eventually, sequentially according to the direction of movement [Ghe91].</td>
</tr>
</tbody>
</table>
Figure 5.1.1 The data flow diagram of the elevator system
Figure 5.1.2 The state transition diagram for the elevator movement

Figure 5.1.3 The Petri net for the elevator upward movement
5.3.2 Convert Each Representation into Internal Form

Initially, the graphical notation of each input model is converted into an internal representative format and stored in the knowledge base. Table 5.2.1 shows the internal format for the data flow diagram. Each component has its own identification number and description. If a component does not have a description in its original model, it is null in the internal format. For example, \texttt{process2(id,[process\_name],[function\_name])} denotes a process in the level 02 data flow diagram. Other models have similar internal formats. The internal formats for the Petri net and the state transition diagram are illustrated in Tables 5.2.2 and 5.2.3, respectively. The internal representation of the Petri net in Figure 5.1.3 is given in Table 5.2.4. If any component in a graphical representation of each input model is not fully described, it needs to be specified clearly in the internal representation. For example, transitions and places in the Petri net in Figure 5.1.3 are denoted as abbreviated symbols but the internal representation in Table 5.2.4 has a full description for each component. This form of internal representation provides the ability to deal with incomplete information. In [Rad91], a CASE tool that converts the text form of a data flow diagram into the internal format needed for this methodology is described.

\textbf{Table 5.2.1} Data flow diagram's internal format

\begin{verbatim}
source2(id,[source\_name]).
sink2(id,[sink\_name]).
process2(id,[process\_name],[function\_name]*).
dst2(id,[datastore\_name]).
dfw2(id,[dataflow\_name]).
\end{verbatim}

Note: \texttt{[function\_name]} := \texttt{[action,object]}
Table 5.2.2 Petri net's internal format

| tran(id, tid, [transition_name]).  
| place(id, pid, [place_name]).  
| arc(id, [pid1, pid2], [tid1], [pid3, pid4]).  
| pconnect(id, did, [description]).  |

Note:

id = {id for each Petri net}, tid = {id for the transition}
pid = {id for the place}
did = {id for the corresponding data flow diagram's component}

Table 5.2.3 State transition diagram's internal format

| state(id, sid, [state_name]).  
| input(id, iid, sid1, sid2, [input_name]).  
| sconnect(id, did, [description]).  |

Note:

id = {id for each state transition diagram}, sid = {id for the state}
iid = {id for each input}
did = {id for the corresponding data flow diagram's component}

Table 5.2.4 The internal representation of a Petri net

tran(1, t_01, ['set request for floor(K+1)']). tran(1, t_02, ['set request for floor(K+1)']).
tran(1, t_03, ['set request for floor(K+1)']). tran(1, t_04, ['set request for floor(L)']).
tran(1, t_05, ['set request for floor(L)']). tran(1, t_06, ['set request for floor(L)']).
tran(1, t_07, ['move up for floor(K+1)']). tran(1, t_08, ['move up for floor(L)']).
tran(1, t_09, ['stop at floor(K+1)']). tran(1, t_10, ['stop at floor(L)']).

place(1, p_01, ['In-button(K+1)=ON']). place(1, p_02, ['Up-button(K+1)=ON']).
place(1, p_03, ['Down-button(K+1)=ON']). place(1, p_04, ['In-button(L)=ON']).
place(1, p_05, ['Up-button(L)=ON']). place(1, p_06, ['Down-button(L)=ON']).
place(1, p_07, ['Currentfloor(K)=ON']). place(1, p_08, ['Requestedfloor(K+1)=ON']).
place(1, p_09, ['Requestedfloor(L)=ON']). place(1, p_10, ['Arrivingfloor(K+1)=ON']).
place(1, p_11, ['Stayingfloor(K+1)=ON']). place(1, p_12, ['Stayingfloor(L)=ON']).

% connection between transitions and places
arc(1, [p_01], [t_01], [p_08]). arc(1, [p_02], [t_02], [p_08]).
ar(1, [p_03], [t_03], [p_08]). arc(1, [p_04], [t_04], [p_09]).
ar(1, [p_05], [t_05], [p_09]). arc(1, [p_06], [t_06], [p_09]).
ar(1, [p_07, p_08], [t_07], [p_08, p_10]). arc(1, [p_07, p_19], [t_08], [p_09, p_10]).
ar(1, [p_08, p_10], [t_09], [p_11]). arc(1, [p_09, p_10], [t_10], [p_12]).
pconnect(1, p_36, [elevator]). % this Petri net is related to process 3.6
Early validation of requirements is important because detection of errors in the early phase definitely reduces software development costs. Each model's internal form is validated according to pre-defined rules to help to detect inconsistent information. If inconsistent facts are found, the information is modified and reentered. The evaluation process for a data flow diagram information is assisted by a regenerated informal English document [Lee91]. The Petri net can be validated by using the formal properties of the Petri nets, such as a reachability tree [Pet81]. By counting the number of token at the particular places, undesirable execution of the Petri nets can also be detected.

5.3.3 Build the Knowledge Base

The next step is to build a knowledge base. It consists of the internal information of the input models, defined rules, and the information derived by the rules. A set of rules derives additional information from initial information. The rules for extracting information from each input model are discussed in Section 5.3.3.1 thru Section 5.3.3.3. Rules for integration of information and generation of a specification model are introduced in Section 5.3.4 and Section 5.3.5, respectively. As each input model represents a different viewpoint, we obtain different types of information from different models. While the data flow diagrams are introduced to extract the objects, actions and their relationship, the state transition diagrams and the Petri nets are used to extract detailed internal behavior of objects and/or actions.

5.3.3.1 Information from the Data Flow Diagram

The data flow diagram is the main source of objects and actions. The frame of the generated specification model is based on the information in the data flow diagram.
The method which uses only a set of data flow diagrams to build an architectural view of object-oriented systems is introduced in Chapter 4, [Lee90] and [Lee91a]. This methodology builds on that strategy. In Figure 5.2, an object specification module derived from a data flow diagram is introduced. This format is extended with additional information from the state transition diagrams and the Petri nets.

ObjectModule :: [in_button]

Definition is

type : passive
class :
visible : [Sensor]
variable :

method turn on ( );
method turn off ( );

End

Figure 5.2 Frame of an object module

5.3.3.2 Information from the State Transition Diagram

From the state transition diagrams, the states and events of all or part of the system are extracted. The extraction of information from the state transition diagram is straightforward as states and input symbols correspond to conditions and events, respectively. The state transition diagram which describes the behavior of process 3.6 in the data flow diagram is given in Figure 5.1.2, i.e., a state transition diagram which represents the behavior of the elevator movement. The information extracted from this representation is shown in the final object module in Figure 5.4.
5.3.3.3 Information from the Petri Nets

A set of Petri nets that specifies each component (object or action) is used to identify behavior of objects. The Petri nets naturally contain properties of distributed systems. In addition to nondeterministic execution and communication aspects, dynamic behavior of the objects and/or actions can be extracted from the Petri nets.

The interpretation of the Petri nets is similar to that of the state transition diagram. The places and transitions correspond to conditions and events/actions, respectively. To get a clear interpretation, each place and transition needs to be specified with an explicit description. Since we use a set of Petri nets in which each Petri net represents a component (an object or an action), specifying the communication aspects between objects from the Petri nets is difficult from the Petri nets. Thus, we extract the communication routines from the data flow diagram information. In Section 5.3.3.1, the visibility of each object is specified. We regard that an object can communicate only with its visible objects by message passing. From Figure 5.2, an object, \([in\_button]\), is visible by \([Sensor]\). Therefore, \([Sensor]\) and \([in\_button]\) communicate with each other. A part of the example system, upward movement of an elevator is specified with a Petri net in Figure 5.1.3 and its internal representation is introduced in Table 5.2.4.

5.3.4 Synthesize the Input

Integration of the information of the three input models is the critical step in this methodology. A frame model, mainly extracted from the data flow diagram, is constructed with a set of objects and the primary actions. This frame is the definition part of the module. The identification of active type objects is very important, as we regard that only active object modules can execute concurrently by message passing.
After the objects and actions are extracted from the data flow diagrams, the detailed behavior is specified by the state transition diagrams or the Petri nets. For flexibility, we do not require both the state transition diagrams and the Petri nets. The internal behavior and properties in the body part of each module are specified with a generic format.

Integration rules are as follows. The integration process is performed with identification numbers because each model's component is denoted by its own identification number. Since the data flow diagram represents the overall view of a system, information from the data flow diagram becomes a framework for other models. Thus the internal representations for the Petri nets and the state transition diagram must contain extra facts which indicate the relationship between them and the components of data flow diagram. Moreover, internal representation of each Petri net and each state transition should have two identification slots; one for itself and one for the relationship with the data flow diagram component. For example, the fact, $\text{pconnect}(1,p_{36},\text{elevator})$, in Table 5.2.4 is introduced to show that the Petri net which has identification '1' is related to the process 3.6 in the data flow diagram in Figure 5.1.1. The connectivity between the data flow diagram and the state transition diagram is specified in the same way.

5.3.5 Generate an Object-Oriented Specification

The general object module format is shown in Figure 5.3. This generic object specification model follows DOSL syntax and consists of two parts: definition and body. In the definition part, the environment and the frame of the object module are specified. The informal requirements document, which is shown in Section 5.3.1, can be used as comments to help the user. The information from the data flow diagrams is
used to specify the definition part. In the body, the internal behavior of each object is described. The body of an object module is specified with the information from the Petri nets and the state transition diagrams. The internal behavior of an object is specified with the DOSL-like format.

\[
\text{ObjectModule} :: [\text{object\_name}]
\]

**Definition is**

- **type**: \{passive, active\}
- **class**: \{parent objects\}
- **visible**: \{visible objects\}
- **variable**: \{data items\}

\[
\text{method action1 ()}; \quad \text{method action2 ()}; \quad \ldots .
\]

**Body is**

\[
(=> [:\text{method}] \text{ begin}
\]
\[
\ldots \ldots
\]
\[
\text{end}
\]
\[
(=> [:\text{method}] \text{ begin}
\]
\[
\ldots \ldots
\]
\[
\text{end}
\]
\[
\ldots \ldots
\]

**End**

*Figure 5.3* The general format of an object module
In Figure 5.4, a list of the active objects and passive objects and one of the object modules derived from the elevator problem are illustrated. The final object specification consists of a set of passive and active object modules which are derived in the same manner. An active object module acts as a monitor which controls the execution of the passive object modules and executes in parallel with other active object modules.

%%%%%%%% List of extracted objects.

Active Objects : [Sensor] [Scheduler] [Manager]
Passive Objects : [door] [in.button] [ex.button] [elevator] [request] [elevator_DB]

ObjectModule :: [elevator]
   Definition is
      type : passive
      class :
      visible : [Scheduler] [Manager]
      variable :

   Methods
      method select ();
      method move-up ();
      method move-down ();
      method stay ();

   Constraints
      ⊙

   Body is

(⇒ [select ( )])
   :-- The scheduler selects an available elevator
   :-- Petri net representation is not included

(⇒ [move-up(Button,State)] begin
   :-- requested floor button's information is coming.

Figure 5.4 An object module
repeat: ((In-button(K+1) =ON or Up-button(K+1) =ON or 
    Down-button(K+1) =ON and State=ANY) 
    ---->
    ;; set request for floor(K+1) 
    Requestedfloor(K+1)=ON; State=ANY;
    []

((In-button(L) =ON or Up-button(L) =ON or 
    Down-button(L) =ON) and State=ANY) 
    ---->
    ;; set request for floor(L) 
    Requestedfloor(L)=ON; State=ANY;
    []

((Requestedfloor(K+1)=ON and Currentfloor(K)=ON) and State=ANY) 
    ---->
    ;; move up for floor(K+1) 
    Requestedfloor(K+1)=ON; Arrivingfloor(K+1)=ON; State=MOVEUP;
    []

((Requestedfloor(L)=ON and Currentfloor(K)=ON and State=ANY) 
    ---->
    ;; move up for floor(L) 
    Requestedfloor(L)=ON; Arrivingfloor(K+1)=ON; State=MOVEUP;
    []

((Requestedfloor(K+1)=ON and Arrivingfloor(K+1)=ON) State=MOVEUP) 
    ---->
    ;; stop at floor(K+1) 
    Stayingfloor(K+1)=ON; State=STANDBY;
    [ANY] <= [:done (State,Stayfloor(K+1))];
    []

((Requestedfloor(L)=ON and Arrivingfloor(K+1)=ON) and State=MOVEUP) 
    ---->
    ;; stop at floor(L) 
    Stayingfloor(L)=ON; State=STANDBY;
    [ANY] <= [:done (State,Stayfloor(L))];
] end

(=> [:move-down(Button,State)])
    ;; similar to "move-up"
    ;; Petri net representation is not included

(=> [:move-down(Button,State)])
    ;; similar to "move-up"
    ;; Petri net representation is not included

End.

Figure 5.4 (continued)
5.4 Summary

An object-oriented specification methodology that consists of analyzing requirements from multiple modeling formats and integrating them into a high level specification model of distributed systems based on an object-oriented perspective is presented. There is a general lack of supporting tools and methodologies to assist the specifier with the assistance for writing formal specifications in distributed systems. We have developed a methodology which helps to address this problem by providing automated support that has the potential to provide assistance for large-scale software development. The methodology provides flexibility as it does not require all three input models but has the capability to integrate all three models.
Chapter 6

Summary

Methodologies that result in more reliable software systems are clearly needed in software development. The need becomes more serious as systems increase in size and complexity. A variety of methods and methodologies are evolving. In general the initial difficulty in software system development concerns proper modeling of the required system and the specification of the requirements.

Distributed systems are clearly important owing to the powerful computing capacity and the advanced hardware support. However, distributed systems, with the added complexity of the communication and synchronization features, are more difficult to formally specify than are sequential systems. The need for formal methods to specify their behaviors and properties is clear.

6.1 Summary of Results

The goal of this research was to develop a methodology, including a specification language, which aids in the analysis and specification phase of the development of distributed systems. The modularization technique in object-oriented approaches is inherently suitable for the development of distributed systems. By accepting the graphical representations of multiple modeling techniques as input, the methodology generates a specification of distributed systems from an object perspective.

A specification language (DOSL) which has a concise syntactic format, has been designed. DOSL includes the required features for distributed object-oriented systems, such as message passing, data abstraction, concurrency, and nondeterministic execution pattern. Temporal operators are used as prefixes in message passing
statements so that the communication patterns in the message passing between object modules are specified clearly. Each message, which consists of an operation name and a set of optional parameters, has an assigned (or default) priority. Nondeterministic execution within an object module is specified in terms of guarded commands.

A language is formally defined by formal semantics techniques. We employed two operational semantics methods, a transition system method and Petri net method, to provide the formal definition of the meaning of DOSL. The operational semantics defines a language by executing a program on an abstract machine and the meaning of a program is interpreted by a sequence of program state which is changeable during the execution. The Plotkin-style semantics method is widely used to formally define CSP-like languages since this method can define the nondeterministic execution pattern and the communication between processes. The meaning of DOSL is given by this Plotkin-style semantics method, called a transition system method. We also provided the Petri net semantics which graphically define the meaning of each statement in DOSL. In addition, another technique of semantics of the message passing statements is given in terms of predicates. This semantics method is presented to supplement the operational semantics of DOSL and to define explicitly the underlying message passing mechanism.

The methodology, shown in Figure 6.1, assists with the construction of a DOSL specification. It is an integrated, formalized methodology for identification of objects, object properties and object behaviors from multi-model formats such as data flow diagrams, state transition diagrams, and Petri nets. It addresses the extraction of objects, actions and relationships from the problem domain with emphasis on specification of the characteristics of distributed systems. This object identification methodology is supported by a knowledge base that provides for automated analysis and
reasoning about objects and their relationships. The final object model is represented in DOSL, providing a formal mechanism for representing the object information. It also provides constructs that allow for refinement of the specification.

Figure 6.1 The overview of the methodology

6.2 Significance of Results

This research addresses two critical problems in the area of formal specification of distributed computing software systems. The first problem is the need for specification languages that have the constructs required to represent the domain of distributed
systems. Language support, including specification languages for distributed systems is not sufficient. Black stated [Bla87], "we believe that the complexity of distributed applications is heightened by the lack of programming language support of distribution". The second problem is the general lack of supporting tools to assist the specifier with writing formal specifications for distributed systems [Geh85], [Avr86], [Est86], [Nor91]. Avrunin [Avr86] emphasizes this, "designing any concurrent software system, particularly a distributed system, is a complex and error-prone task. .... To overcome these problems, designers need both suitably precise notations for describing system designs and their properties, and also methods for rigorously analyzing the behavior of the system represented by a design".

This research addresses these two problems. First, it addresses the need for a comprehensive distributed specification language by formally defining both the syntax and the semantics of a distributed object-based specification language. The specification language is significant because it provides a formal bridge between the requirements specification, design and implementation phases of distributed object-oriented software development. DOSL has a hybrid format which combines the algebraic approach and the model-oriented approach; the definition part follows the algebraic approach and the body part follows the model-oriented approach. The adoption of temporal logic and temporal operators into the design of a language is another defining feature. We use the temporal operators and temporal logic expressions to explicitly specify communication patterns and constraints of the system, respectively. Since temporal logic has power to express the changes of conditions according to time with simple symbols, use of these notations provides a clear and concise syntax for DOSL. With the temporal logic property, DOSL can also be used for the specifying real-time systems. Another contribution of DOSL is that it includes powerful message passing
statements. Many specification languages lack the explicit expressive feature for mes­sage passing. In addition, DOSL provides the priority in the message.

The semantics of DOSL is formally defined by a Plotkin-style transition system method and the Petri net method which are useful for the definition of distributed pro­gramming languages. Although Plotkin’s transition system does not explicitly define an abstract machine, it shows the details of the execution of program structures with state transitions, that is, the meaning of each command in DOSL is defined in terms of the transition relation which illustrates the change of state and the remaining program structure to be executed. The prominent feature of the transition system method is that it enables to show the detailed executions and state transition steps clearly without executing a program on an abstract machine. The Petri net semantics, which provides a graphical representation that more explicitly shows the distributed features of DOSL, is defined based on the transition system method. The Petri net representations of program structures not only defines the meaning of DOSL but also help to analyze the DOSL specification by using the Petri net property. We also define the underlying constraints of communication methods in terms of predicates that are temporal-logic-like formulae. To fully define the underlying meaning of only the message passing statements, a semantics method which is close to the algebraic specifica­tion technique is used. Unlike the other two methods, this method enables to define the definition of time-dependent changes.

This research also addresses the second problem, the need for tool support, by providing an environment that includes a partially automated methodology for specifi­cation of distributed systems. Automated assistance helps to save time and to elimi­nate inherent mistakes that happen during the manual process. By the establish­ment of a knowledge base, the methodology becomes a semi-automatic methodology and
thus potentially provides support for large-scale software development. Although a fully automated methodology would provide many benefits over the manual manner methodology, it still remains as a future goal of software engineering. Therefore, a user interaction process is necessary. In addition, our methodology contains a method to integrate the different models of the system. In the general case, multiple modeling techniques are used in order to represent a system from different viewpoints. However, for the design of the system, there is a need for a methodology which can combine the requirements, in a well-defined manner, from the different models to produce an integrated specification.

Yet another contribution of this methodology is that it is useful as an assistant for the novice system specifier who is not familiar with the object-oriented development techniques. The benefits of object-oriented approaches are widely known, but the development of software system by these approaches is not yet common. One reason is that system specifiers are not familiar with these new techniques and another reason is that it is difficult to identify the objects and their operations from the initial user requirements. Therefore, this methodology is useful for converting functional-oriented representations into an object-oriented representation [Lee90], [Lee91a], [Lee91b], [Lee91c], [Lee92] and [Car92].

The research results in the following benefits.

• DOSL includes powerful features for specifying distributed systems. In particular it can explicitly specify the communication method in the message passing statements. The most powerful feature in DOSL is its repertoire of message passing statements. Two message passing methods, synchronous and asynchronous, can be explicitly specified in a DOSL specification. The nondeterministic
execution within an object module is specified in terms of guarded commands. In addition, a priority can be assigned to each message to increase efficiency of the execution.

- The semantics of DOSL is defined by two operational semantic approaches. In particular, the Petri net semantics enables not only to define the meaning of the specification but also to analyze the specification.

- Due to its concise syntactic format, the DOSL specification is readable and understandable. Even though DOSL is not executable, it has a clear syntax which has potential for modification. Current research is ongoing to define an executable subset language.

- The methodology provides semi-automated assistance. Deriving a specification is a tiresome task and requires repetition and refinement until a satisfactory specification is constructed. The user requirements are frequently changed and modified. Automated assistance helps to save time and to eliminate inherent mistakes that may arise in the manual process.

- The use of a knowledge base system provides advantages for system development. First, the input representation can be converted into a formalized representation by the use of the knowledge base. Thus, the validation of input model's internal information is possible. Second, the automatic processing in the methodology is possible with the help of the knowledge base. Third, manipulation of the information within the knowledge base is easier. Since there is a high possibility of changes of the requirements of the system, ease of modification within the knowledge base provides many benefits.
• Flexibility is another benefit. The methodology does not require all three input models but it has the capability to integrate all the three. The frame of the specification model can be derived using the data flow diagrams. The other models can prescribe dynamic behavior of the system.

• The methodology has the potential for extension. Other models, such as entity-relationship diagrams, could be included as input.

The integrity of DOSL was accomplished by the development of the formal semantics to ensure that the language is well-defined and free of ambiguity. The design criteria, as stated in Chapter 2, was adhered to rigorously during the language design. Comparison of this language with other existing distributed specification language from the literature confirmed that DOSL makes a unique contribution in that it provides 1) various message passing statements, 2) priority in the message, 3) a concise and readable syntactic format, 4) temporal operators for explicitly representing message passing methods, and 5) a hybrid structural format.

The methodology was applied initially to small-scale problems which are found in the literature. This methodology was applied to a combination of input models. A manual application of the methodology was done and the results were evaluated for correctness, reliability, completeness, consistency, and accuracy. We also used the same application by inputting it into the knowledge base system and compared the results with the manual results. In addition, the automated translation of the results to DOSL was evaluated for accuracy against a manual translation process. The methodology was applied to more complex requirements after utilizing it for small-scale problems and was found to agree with the results of a manual specification of the same problems.
6.3 Future Research

This research serves as a formal basis for continued study of the specification of distributed systems. Specific extensions of this research include:

- Extend the methodology so that it can accept more input models. There exist many other modeling techniques which represent a system pictorially or textually. To enrich the produced specification model, other input models such as an entity-relationship model can be included as input.

- Provide more interactive steps in the methodology so that the user can modify the information in a knowledge base whenever errors or mistakes are found during the process. In addition, more rules need to be defined and added in the knowledge base to verify the correctness of input models.

- To increase the quality of the specification produced, the requirements of the desired system should be fully represented in the input models. Thus, the existing modeling techniques may be modified to provide more information. A variety of different versions of each model should be investigated.

- Expand the specification language so that it includes the features of real-time systems. Temporal logic has desirable properties to specify the requirements of real-time systems. Development of real-time systems as distributed systems is a natural approach since there are underlying common factors between two systems.

- Modify the methodology to accept the graphical representations of input models without manual conversion into internal formats.
• Improve the efficiency of the methodology by implementing it on a parallel machine.
Bibliography


[Pnu77] A Pnueli, "The Temporal Logic of Programs", *18th Annual Symposium on


[Rad91] B. Radhakrishnan, "Traditional to Object-Oriented Analysis and Design: A
CASE Tool-Based Approach", Internal Paper, Louisiana State University,


[Ros77] D.T. Ross and K.E. Schoman, "Structured Analysis for Requirements De­

Vol. 4, April 1985, pp.25-35.

Analysis of Ada* Tasking Behavior", *The Journal of Systems and Software,*
No. 8, 1988, pp.343-359.

Petri net Based Toolkit for Ada Tasking Analysis", *IEEE Trans. on Parallel

1989.

Based on Communicating Petri Nets*, Ph.D dissertation, Department of


[Sto77] J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Pro­

[Sto88] P.D. Stotts, "A Comparative Survey of Concurrent Programming Lan­
guages", N. Gehani and A.D. McGettrick (eds.) in *Concurrent Program­


Vita

Sangbum Lee was born in the suburbs of Taegu, South Korea, on September 16, 1960. He received his B.S. degree in Mechanical Engineering in 1983 from Han Yang University, Seoul, South Korea. He joined the department of Computer Science at the Louisiana State University in 1986 and had received the Master’s degree in Systems Science in the Fall of 1989. Since September of 1990 he has been working as a teaching assistant in the Computer Science Department. His current research interests include distributed systems design, formal specification languages, object-oriented systems development, parallel algorithms and expert systems.
Candidate: Sangbum Lee

Major Field: Computer Science

Title of Dissertation: A Formal Methodology for the Specification of Distributed Systems From an Object Perspective

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

July 15, 1992