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.
A computer-aided protocol design methodology: The production systems approach

Huang, Chung-Ming Dorcy, Ph.D.

The Ohio State University, 1991

Copyright ©1991 by Huang, Chung-Ming Dorcy. All rights reserved.
A COMPUTER-AIDED PROTOCOL DESIGN METHODOLOGY: THE PRODUCTION SYSTEMS APPROACH

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Chung-Ming Huang, B.S., M.S.,

* * * * *

The Ohio State University

1991

Dissertation Committee:
Professor Ming T. Liu
Professor Dik Lun Lee
Professor Mukesh Singhal

Approved by

[Signature]
Adviser
Department of Computer and Information Science
© Copyright by

Chung-Ming Huang, B.S., M.S.

1991
To My Family
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my graduate adviser, Professor Ming T. Liu, for introducing me to the research area of Protocol Engineering and his guidance throughout my Ph.D. study. I also would like to express my special thanks to Professor Dik L. Lee and Professor Mukesh Singhal for serving on my reading committee and for reviewing the manuscript.

I would like to express my deep appreciation to my parents, my brother and my sisters. They have always been encouraging and supportive during my graduate study at The Ohio State University. The greatest thanks and credits are due to my lovely wife, Ye-In Chang, whose encouragement and patience are always my strength in this work.

Research reported herein was supported by U. S. Army CECOM, Ft. Monmouth, NJ, under Contract No. DAAB07-88-K-A003. The views, opinions, and/or findings contained in this paper are those of the author and should not be constructed as an official Department of the Army position, policy or decision. My adviser and I would like to express our thanks to Dr. Charles J. Graff for funding the project and for his encouragement and support.
VITA

December 17, 1961

Born – Miaoli, Taiwan, Republic of China

September 1980 – June 1984

B.S., Department of
Electrical Engineering,
National Taiwan University,
Taipei, Taiwan, Republic of China

September 1986 – December 1987

M.S., Department of
Computer and Information Science,
The Ohio State University,
Columbus, Ohio

January 1988 – present

Ph.D. student, Department of
Computer and Information Science,
The Ohio State University,
Columbus, Ohio


PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Architecture and Networking:
Dr. Ming T. Liu

Studies in Parallel Computing:
Dr. Prasad Vishnubhotla

Studies in Distributed Database Systems:
Dr. Mukesh Singhal
# Table of Contents

DEDICATION .......................................................... ii
ACKNOWLEDGMENTS ........................................... iii
VITA ........................................................................ iv
LIST OF FIGURES ................................................ ix
LIST OF TABLES ................................................... xi

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>INTRODUCTION ............................................... 1</td>
<td></td>
</tr>
<tr>
<td>1.1 Motivations ............................................. 1</td>
<td></td>
</tr>
<tr>
<td>1.2 Previous Work .......................................... 3</td>
<td></td>
</tr>
<tr>
<td>1.3 Our Approach ............................................ 4</td>
<td></td>
</tr>
<tr>
<td>1.4 Major Contributions ................................. 5</td>
<td></td>
</tr>
<tr>
<td>1.5 Organization of the Dissertation .................... 7</td>
<td></td>
</tr>
<tr>
<td>II</td>
<td></td>
</tr>
<tr>
<td>THE OPS5 PRODUCTION SYSTEM ............................ 10</td>
<td></td>
</tr>
<tr>
<td>2.1 The Architecture and Notion ...................... 10</td>
<td></td>
</tr>
<tr>
<td>2.2 The Conflict Resolution Strategy ................. 13</td>
<td></td>
</tr>
<tr>
<td>2.3 External Procedure Calls ............................ 13</td>
<td></td>
</tr>
<tr>
<td>2.4 The RETE Algorithm ................................... 14</td>
<td></td>
</tr>
<tr>
<td>III</td>
<td></td>
</tr>
<tr>
<td>PROTOCOL SPECIFICATION USING PRODUCTION SYSTEMS</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Protocol Specification ............................... 17</td>
<td></td>
</tr>
<tr>
<td>3.2 Protocol Specification Using Production Rules .... 23</td>
<td></td>
</tr>
<tr>
<td>3.3 Discussion ................................................ 28</td>
<td></td>
</tr>
</tbody>
</table>
B.1 The Options Provided by PDPS in the Beginning .................................................. 139
B.2 The Options Provided by PDPS after Validation: Listing all Rules .................. 140
B.3 The Options Provided by PDPS after Validation: Listing all States ................. 142
B.4 The Options Provided by PDPS after Validation: Listing all Errors ................ 148
B.5 The Options Provided by PDPS after Validation: Listing a State ................. 150
B.6 The Options Provided by PDPS after Validation: Listing the Path to a State Using State IDs ................................................................. 151
B.7 The Options Provided by PDPS after Validation: Listing the Path to a State Using States ................................................................. 152
B.8 An Example of the Incremental Validation: Adding a Rule ......................... 154
B.9 An Example of the Execution of PDPS after Adding a New Rule ................. 156
B.10 An Example of the Incremental Validation: Deleting a Rule .................... 160
B.11 An Example of the Execution of PDPS after Deleting a Rule .................... 164

C COMPARISON OF THE TRADITIONAL TEST SEQUENCE GENERATION METHOD WITH THE INCREMENTAL TEST SEQUENCE GENERATION METHOD .................................................. 165
C.1 Comparison between the UIO Method and the Incremental Method ............. 165
C.2 Formal Modeling of the Incremental Test Sequence Generation for the Half Duplex ABP ................................................................. 168

BIBLIOGRAPHY .................................................................................. 174
# List of Figures

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An example of the RETE network</td>
</tr>
<tr>
<td>2</td>
<td>X.25 communication protocol specified in the CFSM model</td>
</tr>
<tr>
<td>3</td>
<td>The protocol specification environment in the OPS5 production system</td>
</tr>
<tr>
<td>4</td>
<td>An example of global state reachability analysis</td>
</tr>
<tr>
<td>5</td>
<td>Execution steps in the incremental validation analysis: adding a communication rule</td>
</tr>
<tr>
<td>6</td>
<td>Execution steps in the incremental validation analysis: deleting a communication rule</td>
</tr>
<tr>
<td>7</td>
<td>Abstract architectures for protocol execution: (a) the simulated case; (b) the real implementation case</td>
</tr>
<tr>
<td>8</td>
<td>Implementation model</td>
</tr>
<tr>
<td>9</td>
<td>The abstract test architectures: (a) the traditional test approaches; (b) the incremental test approach</td>
</tr>
<tr>
<td>10</td>
<td>The abstract test procedure of the traditional test approaches</td>
</tr>
<tr>
<td>11</td>
<td>The abstract test architecture of the local method</td>
</tr>
<tr>
<td>12</td>
<td>The abstract test architecture of the distributed method</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>13</td>
<td>The abstract test architecture of the coordinated method</td>
</tr>
<tr>
<td>14</td>
<td>The abstract test architecture of the remote method</td>
</tr>
<tr>
<td>15</td>
<td>The abstract test architecture of the ferry control method</td>
</tr>
<tr>
<td>16</td>
<td>The abstract test procedure of the incremental test approach</td>
</tr>
<tr>
<td>17</td>
<td>The execution procedure of the incremental test sequence generation</td>
</tr>
<tr>
<td>18</td>
<td>The prototype of the incremental test architecture for the local method</td>
</tr>
<tr>
<td>19</td>
<td>The prototype of the incremental test architecture for the distributed method</td>
</tr>
<tr>
<td>20</td>
<td>The prototype of the incremental test architecture for the coordinated method</td>
</tr>
<tr>
<td>21</td>
<td>The prototype of the incremental test architecture for the remote method</td>
</tr>
<tr>
<td>22</td>
<td>The prototype of the incremental test architecture for the ferry control method</td>
</tr>
<tr>
<td>23</td>
<td>Overview of the protocol design process using the OPS5 production system approach</td>
</tr>
<tr>
<td>24</td>
<td>The protocol used for illustration in this chapter</td>
</tr>
<tr>
<td>25</td>
<td>The transmitter section of the half duplex ABP specified in CFSM</td>
</tr>
<tr>
<td>26</td>
<td>The optimized test sequences generated for weak conformance testing by using the UIO method</td>
</tr>
<tr>
<td>27</td>
<td>The test sequence generated by using the incremental method</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>2</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>59</td>
</tr>
<tr>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>The time-queue table of validating the X.25 protocol in PDPS when process number equals...</td>
</tr>
<tr>
<td>---</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>9</td>
<td>process number equals 9 60</td>
</tr>
<tr>
<td>10</td>
<td>process number equals 10 60</td>
</tr>
<tr>
<td>11</td>
<td>process number equals 11 61</td>
</tr>
<tr>
<td>12</td>
<td>process number equals 12 61</td>
</tr>
<tr>
<td>13</td>
<td>The optimal time-process table of validating the X.25 protocol in PDPS 61</td>
</tr>
<tr>
<td>14</td>
<td>The unique input/output sequences for the protocol in Figure 25 166</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

1.1 Motivations

Recent rapid advances in computer communication result in a great need for standardization in worldwide information systems. It is for this reason that the International Standard Organization (ISO) has developed a basic architecture for distributed information processing systems, which is called the Open System Interconnection (OSI) reference model [33]. The ISO/OSI provides a common communication architecture framework for the interworking of heterogeneous computer systems in distributed computing systems and computer networks. The interworking between the different components of distributed systems and computer networks is controlled by the protocols used for the communication between the different system components. Being the cornerstone of distributed computing systems and computer networks, communication protocols are sets of rules that govern the interactions and coordinations among the communicating entities in distributed computing systems and computer networks. Based on the ISO/OSI reference model, computer communication is organized as a hierarchy of layers. A hierarchy of layers in computer communication, in turn, leads to the concept of layered communication protocols.
In order to design a communication protocol that is free from logical errors and satisfies the functional correctness, a complex and repeated cycle consisting of re-specification and re-validation is executed until there is no logical error and the functional correctness is satisfied; next, a machine executable code is generated according to the validated specification for the protocol implementation; finally, the real protocol implementation is tested before it is used in computer networks.

To facilitate protocol design, protocol engineers need to use formal techniques, which are based on the Formal Description Techniques (FDTs), in specification, validation, implementation, and testing of communication protocols. When FDTs are considered as pure specification languages, they are used only for formally describing the functional specifications of communication protocols. As a result, protocol specifications need to be translated to other representation mechanisms that can formulate logical errors and logical properties in the validation phase, because FDTs cannot formulate logical errors and logical properties. In addition, protocol specifications need to be translated to other representation mechanisms that can formulate the test events generation process, because FDTs cannot formulate the test events generation process in the test phase. Consequently, a lot of software tools need to be developed in order to achieve, either partially or fully, automation of the protocol validation, protocol implementation, and protocol testing. Since only the functional specification part of communication protocols is considered in the original design issues of FDTs, it becomes nontrivial and time-consuming to develop these software tools. Therefore, if protocol engineers want to maximize the automation and minimize the human
intervention during the protocol design process, and to reduce the effort and work spent in developing various software tools that can be used in the protocol design process, they do need to consider using a single FDT in the protocol design process. With this concern, the goal of FDTs is not only as specification languages but also as programming languages. In this way, not only can FDTs formulate logical errors and logical properties in the validation phase, but they can also formulate the test events generation process in the test phase.

1.2 Previous Work

During the last decade, a lot of Formal Description Techniques (FDTs) have been proposed for developing communication protocols. In the traditional approaches, heterogeneous representation mechanisms are used in each phase. Therefore, most of the existing approaches for protocol design are based on the translation of heterogeneous representation mechanisms between different phases. That is, the formal specification of a communication protocol needs to be translated into the corresponding representation mechanism in the following phases: the validation, the implementation and the test phases. Consequently, a translation is required between protocol specification and protocol validation [77, 98], between protocol specification and protocol implementation [6, 7, 9, 98, 114], and between protocol specification and protocol testing [26, 44, 88, 93].

A protocol design methodology based on the translation process is proposed in [98]. In this method, three different representation mechanisms, NESDEL, EYPA and IDL, are used in the specification, validation, and implementation phases, respec-
tively. Accordingly, three translators are used to translate NESDEL (specification) to EYPA (validation), EYPA (validation) to NESDEL (specification) and NESDEL (specification) to IDL (implementation), respectively.

For protocol testing, translators are also required, such as those used for ESTELLE in [93] and for LOTOS in [44], to translate the formal protocol specification to the derivation of conformance test sequences. In [93], ESTELLE specifications are first transformed in some Normal Form Transitions (NFTs); then, graph representations of the control flow and data flow are established using NFTs; finally, test sequences are generated according to the graph representations. In [44], LOTOS specifications are first expanded by reducing some operators, such as parallel composition; then, a tree which indicates all of the possible execution sequences in the expanded specification are established; finally, test sequences are generated according to protocol functions, such as connection establishment and disconnection request.

As a result, this translation process not only makes protocol design very time-consuming but also requires a lot of work in developing different software tools to handle various phases.

1.3 Our Approach

In reality, communication protocols are rule-based and data-driven systems without a fixed order which the submodules can follow, and computations in communication protocols are mainly symbolic with a few numerical computations. These characteristics fall into the applicable problem domain of the OPS5 production system, because the OPS5 production system uses data-sensitive and unordered rules rather than se-
quential instructions as the basic unit of computation, and because the design of the
OPS5 production system is mainly focused on symbolic computations with only a few
numerical computations [14]. Therefore, the OPS5 production system is appropriate
for those problems, such as modeling communication protocols, whose knowledge to
be programmed naturally occurs in rule forms [14]. Furthermore, based on a globally
shared dataspace (working memory) in which different types and levels of informa­
tion are all represented in a uniform structure (element), the OPS5 production system
integrates both rule-based and procedure-based computation formalisms [14].

Using the integration features of the OPS5 production system, the OPS5 pro­
duction system can be used as a single representation mechanism for protocol design.
Based on the single representation mechanism approach, to realize phase B from phase
A (specification $\rightarrow$ validation, specification $\rightarrow$ implementation, implementation $\rightarrow$
testing), protocol engineers can combine those related elements in phase A, such as el­
ements representing communication rules (state transitions) in protocol specification,
and some additional elements used in phase B, such as elements representing global
states in protocol validation, to generate the corresponding computation formulae,
either rule-based or procedure-based.

1.4 Major Contributions

The major contribution of this research is to propose a computer-aided protocol design
methodology based on a single representation mechanism, the OPS5 production sys­
tem, to formally model all of the four phases communication protocol design. Using
the single representation mechanism approach, a Protocol Design Production System
(PDPS) has been implemented in the Encore Multimax multiprocessor machine to facilitate protocol design. In PDPS, a user-friendly interactive environment is supported with an incremental validation process and a knowledge base is provided to answer all possible questions. The incremental validation process provided in PDPS shows its effectiveness and usefulness when the modification is small. Furthermore, based on the use of a single representation mechanism for protocol design and based on modifying test architectures, an incremental protocol test method is also proposed. By keeping track of the status of tested events, the incremental test process eliminates the possibility of repeated execution of some transitions in the communication protocol under testing.

In the specification phase, the modeling of state transitions, such as the specification of incoming events, the predicates and the corresponding actions, is specified in production rules. A working memory (WM) is used for depository of the specification of state transitions (communication rules) and frequently changed information, such as the current state of a communicating entity and variables recording local or global status.

In the validation phase, based on global state reachability analysis [115], transitions between global states, logical errors and logical properties are specified in production rules. The WM is the depository of global states and other frequently changed information used in the global state reachability analysis. Moreover, using the WM to store the validation history, the incremental validation process [65] shows its effectiveness and usefulness when the modification is small.
In the implementation phase, the machine-dependent part is realized using the procedure-based computation formalism. The machine-dependent part includes the procedures used for invoking and detecting events, the procedures used for memory management, and the procedures used for encoding and decoding various protocol data units (PDU) for interlayer communication. The WM is the depository of the abstract representation (abstract primitives) of incoming and outgoing events. The inter-communication between the machine-independent and the machine-dependent parts is achieved by the insertion/extraction elements that represent the corresponding abstract primitives into/from the WM.

In the test phase, the environment-independent test sequence generation is specified in production rules; the environment-dependent inter-communication between the test sequence generation module and external testers is realized using the procedure-based computation formalism. The WM is the depository of the test events, the test history, and the related information used in the test process. By modifying test architectures and by using the WM to keep track of the status of test events, the incremental test process can be supported.

1.5 Organization of the Dissertation

This dissertation is mainly concerned with a computer-aided protocol design methodology using the OPS5 production system. In addition, an incremental validation process and an incremental test process are also proposed.

In Chapter II, one of the powerful production systems that is used in this research, the OPS5 production system, is briefly introduced. First, the architecture and notion
are briefly reviewed. Next, the conflict resolution strategy is explained. Then, the feature of external procedure call is presented. Finally, the RETE algorithm, which is used for resolving the pattern match bottleneck in production systems, is briefly discussed.

In Chapter III, a formal method for protocol specification is presented. First, the requirements of formal protocol specification are briefly described. Next, the formal modeling of communication protocols using the OPS5 production system is explained. This includes how to represent the modeling of transitions and how to specify the state transitions (communication rules). Then, a discussion on the formal representation capability of the OPS5 production system is given.

In Chapter IV, a formal method for protocol validation, which is based on the global state reachability analysis, is presented. First, protocol validation is briefly described. Next, the formal representation of global states and the formal modeling of transitions between global states are explained. Then, logical errors and logical properties that exist in communication protocols and their corresponding formal representations in production rules are introduced. After that, an incremental protocol validation algorithm is described. The usage and the performance of Protocol Design Production System (PDPS), which is executed in the Encore Multimax shared memory multiprocessor machine, is exemplified by validating the X.25 protocol in the following section. Finally, a brief discussion on protocol validation is given.

In Chapter V, a formal method for protocol implementation is presented. First, protocol implementation is briefly described. Next, the general implementation model
using the OPS5 production system is explained. Then, a discussion on the implementation capability of the OPS5 production system is given.

In Chapter VI, a formal method for protocol testing is presented. First, the rationale for the incremental protocol test method is explained. Next, a brief survey of traditional test sequence generation methods is described. Then, a brief survey of traditional test architectures is given. After that, the incremental test sequence generation method and its formal modeling in production rules are introduced. The incremental test architectures are introduced in the following section. Finally, a brief discussion on the incremental test method is given.

In Chapter VII, a summary of the proposed computer-aided protocol design methodology, including both the incremental validation process and the incremental test process, is described. After the summary, some possible extensions to PDPS and some future research issues are presented.

Three appendices are included in the dissertation. In Appendix A, a formal specification of the full-duplex Alternating Bit Protocol is given in OPS5 production rules. Examples of using the Protocol Design Production System (PDPS) are given in Appendix B. A comparison of the traditional test sequence generation method with the proposed incremental test sequence generation method is given in Appendix C.

The dissertation ends with a bibliography, listing 132 works related to the research.
CHAPTER II

THE OPS5 PRODUCTION SYSTEM

This chapter is organized in four sections. The architecture and notion of the OPS5 production system is briefly introduced in Section 2.1. The conflict resolution strategy used in the OPS5 production system is explained in Section 2.2. The feature of external procedure calls in the OPS5 production system is discussed in Section 2.3. Finally, the RETE algorithm, which is used for resolving the pattern match bottleneck in the OPS5 production system, is presented in Section 2.4.

2.1 The Architecture and Notion

Being one of the powerful production-system languages, OPS5 has been implemented in BLISS, LISP [14] and C [1, 45, 46, 47, 60]. It is composed of three components [14, 15, 35, 59]:

1. A set of rules (stored in the production memory).

2. A database (working memory).

3. An interpreter (inference engine).

A rule consists of a precondition-action pair. The left hand side (LHS) of a rule is the preconditions that determine the applicability of this rule. The right hand side
(RHS) of a rule is the actions that will be executed if this rule is applied. A rule in OPS5 has the following general form:

\[
(p \ <\text{identifier}> \ \\
<\text{condition}1> <\text{condition}2> \ldots <\text{condition}n> \\
\rightarrow \\
<\text{action}1> <\text{action}2> \ldots <\text{action}m>)
\]

Each condition is a triple of "object-attribute-value" and is called an element. An element can have zero or more attributes. Each attribute can be referenced by the attribute identifier or the field identifier. For example, \( (\text{person} \ "\text{sex} M " \text{name} Jack " \text{age} 35) \) is an element that is called person (field 1) with three attributes: sex (field 2), name (field 3), and age (field 4); the values of these attributes are M, Jack, and 35, respectively. Conditional elements are simple templates to be matched against data items in the working memory (WM). The values of attributes in conditional elements can be specified as constants, or can be specified by using the pattern operators, including variables, predicates (>, <, =, etc), disjunction, and conjunction. For example, \( (\text{candidate} \ "\text{residence} <\langle\text{Columbus Cleveland}\rangle" \ "\text{sex F} " \text{name} \ <n> "\text{age} \{> 18 < 25\}) \) is a conditional element that has the following conditions:

1. the value of attribute residence can be "Columbus" or "Cleveland" (disjunction);

2. the value of attribute sex should be the constant "F";
3. the value of attribute age should be greater than 18 and less than 25 (conjunction);

4. the value of attribute name is not restricted and is represented by a variable <identifier> (= <n>).

The relationships among conditional elements are represented similarly by using variables in related attributes.

A working memory (WM) is a collection of elements. There are three main actions that can alter the contents of the WM:

1. Make: add a new element.
2. Remove: delete an old element.

An interpreter (inference engine) executes the recognize-act cycle between the execution of the applied rules.

1. Pattern matching: First, the interpreter executes the pattern matching phase to recognize those rules whose LHSs are all satisfied with the current contents of the WM and puts them into a conflict set.

2. Conflict resolution: Then the interpreter executes the conflict resolution phase to select one of these rules from the conflict set according to a predefined control strategy.

3. Action: Finally, the interpreter executes the action part of the selected rule to update the contents of the WM.
2.2 The Conflict Resolution Strategy

The conflict resolution strategy of OPS5 is refractoriness, recency, and specificity [14, 59].

1. **Refractoriness** means that a rule should not be allowed to apply more than once to the same elements in the WM.

2. **Recency** means that, if more than one production rule is applicable, the production rule that matches with the most recently inserted elements has a higher priority to be selected than those with old ones. (This strategy can be regarded as the first-in-last-out stack-like processing, or the depth-first processing.)

3. **Specificity** means that, if more than one applicable production rule matches with the same most recently inserted element, the production rule with more conditions to satisfy has a higher priority to be selected than those rules with fewer conditions, since the former is harder to satisfy than the latter. If there is still more than one production rule that satisfies the above criteria, the system nondeterministically selects any production rule from the applicable ones.

2.3 External Procedure Calls

The OPS5 production system permits external procedure and function calls from the right hand side of any production rule [14]. This facility can enhance the computation capability of the OPS5 production system, especially when some nonsymbolic computations or when computations that will be awkwardly solved in rule-based forms
are included. The format for an external function call is described as follows:

(function name <argument-lists>),

where <argument-lists> is a sequence of zero or more variables (attributes) or constants. A function call will return a value to the corresponding attribute. The format for an external procedure call is described as follows:

(call <procedure name> <interface element>),

All communication between the rule-based computation part and the external procedure is via the interface element, which contains the messages used in the external procedure. In order to manipulate the attributes in an interface element or to insert some elements into the WM from the procedure-based computation side [14, 60], the OPS5 production system also provides commands for use in the called functions and procedures. Depending on the version of the OPS5 production system available, these external functions or procedures can be coded in the implementation languages directly (such as LISP, BLISS or C) or other languages indirectly [14].

2.4 The RETE Algorithm

Pattern matching is usually the bottleneck in rule-based systems. The OPS5 production system improves its capability of pattern matching by using an efficient pattern match algorithm, the RETE algorithm [36], and the C-based implementation [60]. The RETE algorithm creates a data-driven network compiled from the LHSs of all production rules to perform pattern matching effectively. The main characteristics of the RETE algorithm are described as follows:
1. Focus on the update: Inputs to the RETE network are the updates to the WM, and outputs are the changes to the conflict set. Therefore, the whole WM does not have to be scanned in each matching cycle.

2. Test the common expressions only once: This can be divided into the common intra-element feature tests and the common inter-element feature tests. Therefore, redundant tests can be avoided.

3. Incremental matching: The intermediate results are stored in the memory to avoid unnecessary re-tests.

Figure 1 shows an example of a RETE network. The node which has a single predecessor is concerned with one intra-element feature test. The node which has two predecessors is used to check one inter-element feature test.

Currently, a parallel C-based OPS5 using the RETE algorithm is implemented in the Encore Multimax shared memory multiprocessor machine with 16 processors at Carnegie-Mellon University [46, 47, 60]. The parallel implementation has up to 200 fold speed-up over the sequential LISP-based implementation of OPS5 [60]. The version of Encore Multimax available at The Ohio State University has 12 processors, and each of which is connected to the 32 Mbytes shared memory. Therefore, the speed-up which this research can get will be less than 200.
Changes to the WM

- element =global-state?
  - entity=1?
    - size=0?
      - join the elements in which the value of the id attribute from the left is equal to the value of the id attribute from the right
      - report that production quiescent-state is satisfied
  - entity=2?
    - size=0?
      - join the elements in which the value of the state attribute from the left is equal to the value of the state attribute from the right
      - report that production final-state is satisfied

- element =global-state?
  - entity=finalstate1?
    - size=0?
      - join the elements in which the value of the state attribute from the left is equal to the value of the state attribute from the right
      - report that production final-state is satisfied

Figure 1: An example of the RETE network
CHAPTER III

PROTOCOL SPECIFICATION USING PRODUCTION SYSTEMS

This chapter is organized in three sections. The requirements of the formal protocol specification are briefly described in Section 3.1. The formal specification of communication protocols using OPS5 production rules is explained in Section 3.2. A final discussion on the formal specification capability of the OPS5 production system is given in Section 3.3.

3.1 Protocol Specification

The standard protocol specifications are often written in natural languages, since these natural languages are easily understandable by human beings [11]. While protocol specifications written in natural languages are readily accessible to protocol designers, they have the following disadvantages compared to protocol specifications written in formal languages [11]:

1. Protocol specifications using natural languages often contain ambiguities and are difficult to check for completeness and consistency.

2. Protocol specifications using natural languages are not helpful for the automation of protocol validation, implementation, and conformance testing, because
they cannot be processed by automated tools.

The purpose of using formal specification languages is twofold:

1. Formal specification languages provide concise and precise descriptions for communication protocols.

2. The formal nature of protocol specifications makes it possible to apply certain automated tools during the protocol design life cycle.

For layered communication protocols, these formal descriptions include the interactions between a local entity and its peer entity, the interactions between a local entity and the environment, and the operations in response to the service requests invoked from the upper layer. Many formal methods have been used to formally specify communication protocols, such as communicating finite state machines (CFSM) [12], petri nets [34], programming languages [39, 48, 96, 109], and formal grammars [4, 108, 111]. For example, Figure 2 depicts the X.25 packet level DTE/DCE interface communication protocol modeled in CFSM [76], where ‘−’ represents a send transition, ‘+’ represents a receive transition and the circle represents a state of a communication entity. In the recent development of Formal Description Techniques (FDTs), based on the extended state transition model, ESTELLE and SDL were proposed in [130] and [132], respectively; based on the process algebras, LOTOS was proposed in [131].

The specification of a communication protocol should define all the relevant behavior aspects of the specified object, but nothing more. The other aspects should be left undefined, so that the protocol specifications can be generic enough to fit various
Figure 2: X.25 communication protocol specified in the CFSM model
implementation requirements. Therefore, a formal specification of a communication protocol is usually composed of two parts, namely, the machine-independent part and the machine-dependent part. The machine-independent part includes those rules that define the interactions of communication entities in response to incoming events and the interactions in response to changes in the local environment. This part can be specified in the specification phase and a high level of abstraction is needed to permit different realizations for various implementation environments. The machine-dependent part includes the procedures for invoking and detecting events, memory management, inter-task communication, and encoding and decoding various protocol data units (PDUs) for interlayer communication, where PDUs represent the interactions exchanged through the underlying communication service between the peer protocol entities [11]. This part cannot be completely specified until the implementation phase, due to the fact that its realization relies heavily on the operational architecture and the host operating system. Therefore, abstraction mechanisms are needed in the specification phase to describe communication protocols without specifying implementation details and without sacrificing generic realization. Since the machine-independent part is more descriptive and the machine-dependent part is more prescriptive, a rule-based representation formalism is suitable for the machine-independent part and a procedure-based representation formalism is suitable for the machine-dependent part.

To adequately describe communication protocols, a Formal Description Technique (FDT) should satisfy the following requirements [83]:
1. Abstract descriptions: to allow the machine-dependent part to be specified abstractly.

2. Modular descriptions: to increase the readability and the modifiability.

3. Modeling of concurrency: to describe the concurrent executions of communication entities.

4. Modeling of nondeterminism: to represent the nondeterministic behavior of communication entities in choosing transitions.

5. Descriptions of predicates and variables: to support the complex protocols with predicates and variables.

6. Numerical computations and text processing: to allow the processing of variables and message texts.

In order to facilitate other protocol design tools, such as tools for protocol validation, protocol implementation, and protocol testing, the following properties are also required [83]:

1. Executability: to facilitate direct simulation, real implementation, and conformance testing.

2. Expressing functional properties: to facilitate formal validation.

3. Expressing performance properties: providing the time specification and the probability specification of each transition to facilitate performance analysis.
By using the OPS5 production system, communication protocols are modularly represented in production rules. This modular production representation makes the formal specification of a communication protocol very natural to describe and easy to modify. Modeling of concurrency and nondeterminism is embedded in the conditional elements of each production rule. The inference engine triggers the corresponding production rule according to the current configuration in the WM. Variables in protocols can be expressed by attributes in some elements, or by one specific element to record all of the variables. For predicates, they can be modeled by pattern expressions in elements. The satisfaction of a predicate is embedded in the existence of the corresponding element(s) in the WM and/or in the matching with some local or global status recorded in other elements. The processing of message texts and the numerical computation for variables are also supported by the system functions in the OPS5 production system, such as compute and substr (see Section 3.2). The machine-dependent part is abstractly described using external procedure calls. Through the capability of pattern matching, all of the attributes of elements in production rules and in external procedure calls act as data templates to have generic data types. In this way, the formal specification only describes the relations between the before-state, the after-state, and the occurring event(s), but does not specify any implementation details. Figure 3 shows the protocol specification environment in the OPS5 production system [53, 54].
3.2 Protocol Specification Using Production Rules

There are two types of transitions in communication protocols: one is external events with other layers, such as the send and receive transitions; and the other is internal events with the local environment, such as the time-out event. Using the OPS5 production system, transitions of external events can be described in one uniform element rule. For example, the following two elements describe the possible send and receive transitions for entities DTE and DCE, respectively.

(rule "id 1 "entityDTE "type s "cstate1 "message call.request "nstate 2)

(rule "id 2 "entityDCE "type r "cstate1 "message call.request "nstate 2)
The modeling of a state transition can be specified by a production rule. For example, a send transition is modeled by the following production rule:

\[
(p \text{ send\_transition})
\]

\[
(\text{Inform } ^*\text{entity } <e> ^*\text{state } <cs>)
\]

\[
(\text{rule } ^*\text{id } <i> ^*\text{entity } <e> ^*\text{type } s ^*\text{cstate } <cs> ^*\text{message } <m> ^*\text{nstate } <ns>)
\]

\[
\rightarrow
\]

\[
(\text{call transmit DataRequest } ^*\text{entity } <e> ^*\text{message } <m>)
\]

\[
(\text{modify 1 } ^*\text{state } <ns>))
\]

This rule can be applied when entity <e> is in state <cs> and there is a send transition in which entity <e> sends a message <m> from current state <cs> to next state <ns>. In the LHS, element Inform records the current state of the entity. In the RHS, the first action calls external procedure transmit with element DataRequest. Element DataRequest represents the abstract service primitive for data transmission, where service primitives represent the primitive interactions between communication layers. External procedure transmit, in turn, invokes the corresponding system process to transmit the message. The second action updates the entity's state recorded in the first element Inform.

Similarly, a receive transition is modeled as follows:

\[
(p \text{ receive\_transition})
\]

\[
(\text{DataIndication } ^*\text{entity } <e> ^*\text{message } <m>)
\]
(Inform "entity <e> "state <cs> )

(rule "id <i> "entity <e> "type r "cstate <cs> "message <m> "nstate <ns> )

->

(modify 2 "state <ns> )

(remove 1))

where element DataIndication represents the abstract service primitive for data indication. Element DataIndication contains the receive message and is inserted into the WM by the corresponding system process.

Internal events are represented by the insertion of elements or by external procedure calls with the corresponding interface elements. For example, a time-out event for the full duplex Transport Alternating Bit Protocol (ABP) [69] is modeled as follows:

(p Receive_Timeout

(timeout "Aentity <A> "N_Lpoint <N> )

(Inform "Aentity <A> "N_Lpoint <N> "U_Lpoint <U> "state <s1> "Tseq <sn> "Time <t> )

{(Tdata "N_Lpoint <N> "data <m> ) <buf> } )

(commrule "Ipoint server "mess time_out "cstate <s1> "nstate <s2> )

->

(call trandata DataRequest "Aentity <A> "N_Lpoint <N> "time <t> "seq <sn> "data (substr <buf> data inf) )

Attributes `Aentity`, `N.Ipoint` and `U.Ipoint` represent the identifiers of the local communication entity, the network interaction point and the user interaction point, respectively. Element `timeout` representing the abstract primitive of the time-out event is inserted by the system process that administers the timer for message transmission.

Element `Inform` records variables: attribute `Tseq` records the current send sequence number and attribute `Time` indicates the time-out period. Another attribute `Rseq` (which indicates the currently expected receive sequence number) in element `Inform` is not expressed explicitly, because attribute `Rseq` is not referenced in this production rule. Element `Tdata` is the send buffer that stores the currently transmitted data block to be acknowledged. Element `commrule` records the state transition rule for the time-out event. Element `DataRequest` represents the abstract network data request service primitive, where function `(substr <buf> data inf)` extracts values from attribute `data` to the end field (denoted as `inf`) of element `Tdata` (designated by "<buf>"). This rule is applied when local entity `<A>` is in state `<s1>` and receives a time-out event. In the RHS, the production rule calls external procedure `trandata` which, in turn, invokes the corresponding system process to re-issue the time-out transmission message and to initiate a timer to monitor this re-transmission event.

In a communication protocol, predicates are used as the preconditions to trigger a transition and variables are used to record some local or global status. For example, to specify the reception of an abstract network data indication service primitive
which has the expected sequence number in the full duplex Transport Alternating Bit Protocol (ABP) [69], one can use the following production rule:

\[
(p \text{ Receive}_\text{Data} \\
\{(\text{DataIndication } ^\wedge N\text{-Ipoint } ^N > ^\wedge \text{seq } ^\wedge <sn> ^\wedge \text{type } ^\wedge \text{trndata } ^\wedge \text{data } ^\wedge <m>) \\
<\text{buf}> \} \\
(\text{Inform } ^\wedge A\text{entity } ^A > ^\wedge N\text{-Ipoint } ^N > ^\wedge U\text{-Ipoint } ^U > ^\wedge \text{state } ^s1 > ^\wedge Rseq <sn>) \\
(\text{commrule } ^\wedge \text{Ipoint server } ^\wedge \text{mess } ^\wedge R_D ^\wedge c\text{state } ^s1 > ^\wedge n\text{state } ^s2)) \\
\text{-->} \\
(\text{call } ^\wedge \text{tranack } ^\wedge \text{DataRequest } ^A\text{entity } ^A > ^\wedge N\text{-Ipoint } ^N > ^\wedge \text{seq } ^s <sn> \\
^\wedge \text{data } (\text{substr } <\text{buf}> \text{ data inf})) \\
(\text{modify } 2 ^\wedge \text{state } ^s2)) \\
(\text{modify } 2 ^\wedge Rseq (\text{compute } 1 - <sn>)) \\
(\text{remove } 1))
\]

Element \textit{DataIndication} representing the abstract network data indication service primitive is inserted by the system process that administers the receptions from the network layer. Element \textit{DataIndication} records the network interaction point, the sequence number of the message, the message type in this primitive and the data block. To express the predicate that the receive sequence number is equal to the currently expected one, the same identifier is used in the attributes \textit{seq} of the element \textit{DataIndication} and \textit{Rseq} of the element \textit{Inform}. 


This rule is applied when local entity <A> is in state <s1> and receives a data block with the expected sequence number from network interaction point <N>. In the RHS, the production rule calls external procedure tranack which, in turn, invokes the corresponding system process to transmit an acknowledgment to the peer entity and the corresponding system process that administers the receive data to indicate the readiness for upper users' reception. Then, the next expected receive sequence number in the attribute Rseq of the element Inform is calculated by using the function compute. Five arithmetic operators (addition, subtraction, multiplication, division and modulus) used in communication protocols are all supported.

3.3 Discussion

Currently, several shared dataspace languages, such as PROLOG [102, 103] and L.0 (developed at Bellcore) [29], are used only in rapid prototyping for simulation (validation or performance analysis), but are not used for real implementation. The reason is that both PROLOG and L.0 lack the integrated functionality for inter-program communication [13]. In contrast to these shared dataspace languages, OPS5 can be used for both simulation and real implementation. Using the OPS5 production system approach, communication rules (state transitions) are specified as triples of "object-attribute-value" and the modeling of state transitions is specified by production rules. The inference engine in the OPS5 production system acts as the dispatcher to trigger the applicable production rule according to the current configuration of the WM.

Abstract properties of a protocol entity are often described in terms of the exchange of Protocol Data Units (PDUs) with the peer entity and service primitives
with the user. One of the major advantages of the OPS5 rule-based specification is its simple and flexible structure. It uses production rules for functional specifications and uses elements to store different types and levels of information. The pattern match computation allows each attribute to act as a data template for generic data types and allows both high-level and low-level specifications. Therefore, a protocol designer can specify a communication protocol very primitively or in detail. If the protocol designer wants to specify a protocol primitively or to specify a protocol for more generic realization without dealing with the details of PDUs, then each PDU can be shrunk into as few attributes as possible. That is, those fields in a PDU which are not related to the predicates of state transitions can be represented by a single symbolic name (one attribute). On the other hand, if the protocol designer wants to specify a protocol in more detail, then a PDU can be expanded to a number of attributes. That is, each field of a PDU is represented by one attribute. For the abstract specification of interlayer interactions, the abstract service primitives are represented by elements or external procedure calls with the corresponding elements. Each service primitive usually includes several parameters that are exchanged between the service user and the protocol entity during the execution of service primitives. A single attribute can represent the following parameters in a service primitive:

1. Those parameters that are not related to the predicates of state transitions.

2. Those parameters that represent user data which can be transmitted by the communication service to the remote user without interpretation.

The above parameters can also be expanded to a number of attributes, if the
protocol designer wants to specify service primitives in more detail. For illustration, a complete specification of the full duplex Transport Alternating Bit Protocol (ABP) [69] represented in the OPS5 rule-based specification is listed in Appendix One.

The flexibility of OPS5 also allows the incorporation of new features. Since the semantics of each element or each attribute is decided by protocol designers, they can incorporate new features into the system easily without changing the underlined software. For example, the priority description in a communication protocol can be specified as an additional attribute in the corresponding element; i.e., the protocol designer can add the attribute priority to the element rule described in Section 3.2. Furthermore, if the protocol designer wants to specify the control arbitration of the priority-based events, the following three production rules can be added to describe the control arbitration:

1. (p Send_vs_Receive_Competition
   (currentpriority *priority <p> *type <t>)
   (Inform *entity <e> *state <cs>)
   (rule *entity <e> *type s *priority <p1> *cstate <cs>)
   (DataIndication *entity <e> *message <m>)
   (rule *entity <e> *type r *priority {<p2> < <p1> } *cstate <cs>
      *message <m>)
   -*>
   (modify 1 *priority <p1> *type s))

2. (p Receive_vs_Receive_Competition
(currentpriority "priority <p> "type <t> )

(Inform "entity <e> "state <cs>)

(DataIndication "entity <e> "message <m1>)

(rule "entity <e> "type r "priority <p1> "cstate <cs> "message <m1>)

(DataIndication "entity <e> "message {<m2> <> <m1>})

(rule "entity <e> "type r "priority {<p2> > <p1>} "cstate <cs> "message <m2>)

- ->

(modify 1 "priority <p2> "type r))

3. (p Multiple_Transition_Send_Win

(currentpriority "priority <p> "type s)

(Inform "entity <e> "state <cs>)

(rule "id <i> "entity <e> "type s "priority <p> "cstate <cs> "message <m> "nstate <ns>)

- ->

(call transmit DataRequest "entity <e> "message <m>)

(modify 2 "state <ns>)

(modify 1 "priority 0))

Note that the production rule of Multiple_Transition_RECEIVE_Win can be added in the same way as the third production rule, and the action (make currentpriority "priority 0 "type nil) is added to the end of the RHS of each production rule
used for describing the modeling of send and receive transitions. According to the recency conflict resolution strategy, production rules \( X Competition \), where \( X \) is either Send\_vs\_Receive or Receive\_vs\_Receive, will be executed after the execution of an event in order to find the highest priority event among the applicable ones in the WM. Then the winning event is executed by production rule \( MultipleTransition\_Y\_Win \), where \( Y \) is either send or receive. If there is only one event in the WM, production rule \( X Competition \) is not triggered and one of the original production rules for the modeling of send and receive transitions is executed.

One advantage of formal specifications is that they can facilitate the execution of communication protocols for different purposes. In general, based on the formal specifications, there are three different modes of execution [9]:

1. Simulated execution of formal specifications: This mode is used to analyze the logical correctness of the protocol, or to analyze the performance to determine optimal parameters for real implementations. This mode can be realized in a closed way. That is, all of the communication entities can be specified in OPS5 production rules, and the communication medium is represented by some elements in the WM. For the validation of logical correctness, the send and receive transitions, and all of the logical errors and logical properties can be formally modeled in some production rules; the inference of reachable global states is executed by the inference engine [52, 54]. For the performance analysis, one can use the algorithms for timed reachability analysis and performance analysis developed in [67] to formally model communication protocols and can
have performance simulations in the OPS5 production system.

2. Traditional meaning of implementation: The communication protocol is executed to provide a real communication service in a real operational system. In the formal specification, the machine-independent part is formally specified in production rules, the machine-dependent part is specified abstractly or left unspecified. The realization of machine-dependent part, pertaining to the details of the operational architecture and the host operating system, is normally coded in procedure-based computations in the implementation phase. In this way, four dependent modules are created. One deals with the formal specification of protocols in production rules; the other three that are coded in a procedural language deal with the interactions with the upper layer, the lower layer and the local environment, respectively. The inter-module interactions can be realized by insertion/extraction of elements into/from the WM [53, 54].

3. Conformance testing: This is used to test a real implementation for conformance with the original protocol specification. In order to have a higher degree of accuracy and automation, a protocol conformance test system is first constructed to generate the corresponding test events and execute the interactions with Implementation Under Testing (IUT) according to the test events, and is then used to analyze the test results and determine whether the trace of interactions observed during the conformance testing satisfied the requirements of the reference specification. In [55], an incremental test method is proposed using OPS5.
The above three execution modes using the OPS5 production system will be explained in detail in the following three chapters.
CHAPTER IV

PROTOCOL VALIDATION USING PRODUCTION SYSTEMS

This chapter is organized in six sections. A brief introduction of protocol validation is described in Section 4.1. The formal modeling of global states and the formal modeling of the transitions between global states using OPS5 production rules are explained in Section 4.2. Logical errors and logical properties and their formal modeling in OPS5 production rules are described in Section 4.3. The incremental validation algorithm is introduced in Section 4.4. The usage and the performance of Protocol Design Production System (PDPS) are presented in Section 4.5. Finally, a brief discussion on protocol validation is given in Section 4.6.

4.1 Protocol Validation

A validation process has the objective of ensuring that the functional specification corresponds to the requirements and that the specification is internally consistent. The validation process should be done as early as possible in the design phase, because errors in the specification are very expensive to eliminate if they are found during the later phases of the protocol design life cycle. The essence of a protocol is the behavior of the communicating protocol entities, i.e., the set of possible sequences of events
that can occur when the protocol entities are in interaction. Therefore, protocol validation [8, 61, 64, 73, 86, 105, 115] is a procedure to detect logical errors and logical properties, such as deadlock, unspecified receptions, channel overflow (when communication channels are finite), nonexecutable interactions, quiescent states and ambiguous states [78], in communication protocols. Depending on the models used for formal protocol specification, there are two approaches to protocol validation: global state reachability analysis and deductive inference (program proofs) [73]. Global state reachability analysis is based on exhaustively exploring all of the possible interactions of the protocol entities. Deductive inference is based on a list of safety and liveness properties and a list of axioms and rules for inferring the statements from the axioms. Although global state reachability analysis is more simple and more straightforward, and although it is in principle trivial to derive the global state space, it is tedious to do so manually. Fortunately, global state reachability analysis is easily automated. Consequently, global state reachability analysis has attracted much attention during the last decade and probably will continue to be the most widely used method for protocol validation in the future [86]. For this reason, this dissertation uses global state reachability analysis for validating the logical correctness of communication protocols.

In global state reachability analysis, a reachable global state graph containing all possible transition sequences in the communication protocol is generated. This reachable global state graph is rooted from the initial global state in which each communicating entity is in its initial state and each communicating channel is empty.
From the initial global state, global state reachability analysis examines which events can occur and to which state the corresponding transition leads. Then from each of the new global states, global state reachability analysis repeats this procedure until there is no more transition, or all global states are already explored. All of the logical errors and logical properties stated above can be identified in global state reachability analysis. Figure 4 shows an example of global state reachability analysis.

In this dissertation, a computer-aided protocol design system (Protocol Design Production System: PDPS) is developed by using the OPS5 production system and is executed in the Encore Multimax shared memory multiprocessor machine with 12 processors. Using PDPS, the global state space of the set of interconnected protocol entities is calculated and analysed. Based on the OPS5 production system approach, global state reachability analysis can be formally modeled in production rules. Some of the production rules describe the transitions between reachable global states, and the others define logical errors and logical properties. In PDPS, a user-friendly interactive environment is supported with an incremental validation process and a knowledge base is provided to answer all possible questions. Since knowledge is represented in rules and is separated from control in PDPS, protocol designers obtain the advantages of clear knowledge representation, strong modularity, modifiability, and expressibility. Moreover, through the combined use of parallel computing [46, 47, 60] and a fast pattern match algorithm [36], PDPS can render good performance for global state reachability analysis. Furthermore, by using the WM to record the validation history, an incremental validation process to facilitate protocol design
entity 1

entity 2

Figure 4: An example of global state reachability analysis
is also provided in PDPS.

4.2 Modeling of Global States and Modeling of Global State Transitions

A globalstate element is used to describe a global state. For example, the initial global state of the X.25 protocol depicted in Figure 2 can be described as follows:

\[(\text{globalstate "entity DTE "id 1 "size 0 "state ready "queue nil})\]
\[(\text{globalstate "entity DCE "id 1 "size 0 "state ready "queue nil})\]

Since the size of a queue is variable and since only one vector attribute (an attribute with variable size) is allowed in OPS5, two elements are used to describe a global state and are connected together by the same identifier, i.e., \(id\). Note that the queue in the two connected elements represents a channel from the peer entity to the local entity.

Production rules can be used to describe the transitions between reachable global states. The following two production rules can describe all possible transitions from a global state:

1. \((\text{p e1.send_message_to_e2})\)

\[\{(\text{globalstate "entity <e1> "id <i> "size <s1> "state <x>] <h1>\})\]
\[\{(\text{globalstate "entity {<e2> <> <e1>} "id <i> "size <s2> "state <y>] <h2>\})\]

\[(\text{rule "id <r> "entity <e1> "type s "cstate <x> "message <m> "nstate <n>})\]
(bind <j> (genatom))
(bind <s3> (compute <s2> + 1))
(make-globalstate "entity <e1>" "id <j>" "size <sl>" "state <n>" "queue (substr <h1> 6 inf))
(make-globalstate "entity <e2>" "id <j>" "size <s3>" "state <y>" "queue (substr <h2> 6 inf) <m>)
(make-rule-state "ruleid <r>" "stateid <j>"))

2. (pel.receive_message_from.e2
{(globalstate "entity <e1>" "id <i>" "size <s1>" "state <x>" "queue {<m> <>
nil}) <h1>}
{(globalstate "entity {<e2> < > <e1>}" "id <i>" "size <s2>" "state <y>"
<h2>}
(rule "id <r>" "entity <e1>" "type r" "cstate <x>" "message <m>" "nstate
<n>
--> 
(bind <j> (genatom))
(bind <s3> (compute <s1> - 1 ))
(make-globalstate "entity <e1>" "id <j>" "size <s3>" "state <n>" "queue (substr <h1> 7 inf))
(make-globalstate "entity <e2>" "id <j>" "size <s2>" "state <y>" "queue (substr <h2> 6 inf))
(make rule-state `ruleid <r> `stateid <j>))

where bind is an operation to assign the value of the second parameter to the first
parameter and genatom is a function to create a unique number.

The first production rule describes a send transition of entity <e1>. This rule can
be applied when entity <e1> is in state <x> and there is a send rule in which entity
<e1> sends a message <m> to entity <e2> from current state <x> to next state
<n>. Then, a new global state is generated and inserted into the WM and a mark is
set to record that this rule has been used. Actions 3 and 4 copy the original contents
of the queue to the new global state and append the message to the corresponding
queue, respectively.

The second production rule describes a receive transition of entity <e1>. This
rule can be applied when entity <e1> is in state <x>, a message <m> is in the head
of the queue from entity <e2> to entity <e1> and there is a receive rule in which
entity <e1> receives a message <m> from state <x> to state <n>. The remaining
parts are similar to those of the send transition.

4.3 Modeling of Logical Errors and Logical Properties

In the same way, all logical errors and logical properties can be defined in the form
of production rules.

1. Unspecified Reception: An unspecified reception error occurs when an entity is
   in a state such that there is no specified receive transition corresponding to the
   reception of a message which is in the head of the receive channel [78, 115]. The
formal definition in a production rule can be described as follows:

(p reception-error
  (globalstate "entity <el> "id <i> "size {<s1> > 0} "state <x> "queue
   {<m> <> nil})
  - (rule "entity <el> "type r "estate <x> "message <m> "nstate
   <n>)
  - >
  (make reception-error "id <i>)
  (write RECEPTION ERROR IN ENTITY <el>! STATE ID =
   <i>))

where the symbol "-" before the second condition element represents "not exist." The other more restricted definition of the unspecified reception error is limited to a receive state only (in which there is no send transition) [41]. In this case, one more condition element should be added to the precondition part: - (rule "entity <el> "type r "cstate <x>).

2. Deadlock: A deadlock error occurs when all channels are empty and all entities are in the states in which no send transition exists [78, 115]. If the protocol is acyclic, these states should not be final states. In other words, if one of the entities is not in the final state, this global state is in a deadlock state. The formal definition in the form of production rules is described as follows:
(a) (p deadlock1

(globalstate "entity 1 "id <i> "size 0 "state <x>)

(globalstate "entity 2 "id <i> "size 0 "state <y>)

- (rule "entity 1 "type s "cstate <x> "message <m1> "nstate <n1>)

- (rule "entity 2 "type s "cstate <y> "message <m2> "nstate <n2>)

- (globalstate "entity final-state1 "state <x>)

- ->

(make deadlock-error "id <i> )

(write DEADLOCK ERROR ! state ID = <i> ))

(b) (p deadlock2

(globalstate "entity 1 "id <i> "size 0 "state <x>)

(globalstate "entity 2 "id <i> "size 0 "state <y>)

- (rule "entity 1 "type s "cstate <x> "message <m1> "nstate <n1>)

- (rule "entity 2 "type s "cstate <y> "message <m2> "nstate <n2>)

- (globalstate "entity final-state2 "state <y>)

- ->

(make deadlock-error "id <i> )

(write DEADLOCK ERROR ! state ID = <i> ))

3. Channel Overflow: For a communication system whose communication channels are finite, a channel overflow error occurs when a protocol entity attempts to send a message into a channel which has already reached its maximum capacity [78, 115]. The formal definition in the form of a production rule is described as
follows:

(p channeloverflow
(channelbound "number <n>")
(globalstate "entity <el> "id <i> "size {<sl> > <n>})
- >
(make channel_overflow "id <i>")
(write CHANNEL OVERFLOW IN QUEUE TO ENTITY <el> STATE ID = <i>))

4. Nonexecutable Interaction: A nonexecutable interaction is a communication rule which has been specified but has never been executed [78, 115]. It can be found by the following production rule after all global states are generated:

(p redundant_rule
(rule "id <i>")
- (rule-state "ruleid <i> "stateid <s>)
- >
(make redundant_rule "id <i>")
(write REDUNDANT RULE ID = <i>))

5. Quiescent State: A quiescent state is a state in which both communication channels are empty [78]. The formal definition in the form of a production rule is described as follows:
6. Ambiguous State: An ambiguous state is a state in which an entity's state can coexist in more than one different quiescent state [78]. The formal definition in the form of a production rule is described as follows:

\[
(p \text{ ambiguous-state} \\
\text{(globalstate } \text{ entity} <e1> \text{ id } i \text{ size 0 state } x) \\
\text{(globalstate } \text{ entity} \{<e2> <> <e1>\} \text{ id } i \text{ size 0 state } y) \\
\text{(globalstate } \text{ entity} <e1> \text{ id } \{<j> <> <i>\} \text{ size 0 state } \{<z> <> x\}) \\
\text{(globalstate } \text{ entity} <e2> \text{ id } j \text{ size 0 state } y) \\
\rightarrow \\
\text{(make ambiguous-state } \text{ entity} <e1> \text{ id1 } i \text{ id2 } j) \\
\text{(write IT IS AN AMBIGUOUS STATE ID = } i \text{ IN ENTITY } <e1> \\
\text{WITH STATE ID = } j))
\]
4.4 Incremental Validation

In order to design a correct communication protocol, a complex and repeated cycle consisting of re-specification and re-validation is executed. In the non-incremental validation case, any modification of communication protocols will invoke a re-validation process from the beginning. When there are a lot of modifications, including adding or deleting too many communication rules or modifying the data structures that result in many modifications in communication rules, this non-incremental approach is preferable. However, when only a few communication rules are modified, the non-incremental approach makes protocol design very time-consuming. The reason is that those global states that are error-free and are unrelated to the modification still have to be re-explored in the non-incremental validation approach. The incremental validation process shows its effectiveness and usefulness when modification is small. Therefore, an incremental validation process is developed for use in PDPS. In this section, an incremental validation algorithm and its formal representation in production rules are briefly described.

There are two parts in the incremental validation process: adding or deleting communication rules. In both cases, the system will first check whether this communication rule has already existed or not. In the case of adding a communication rule, if this communication rule does exist, then it is a duplicated communication rule. In the case of deleting a communication rule, if this communication rule does not exist, then it is meaningless to delete a non-existing communication rule. The following two production rules are used to check these two cases:
where element \textit{new} (\textit{old}) is a flag to denote the communication rule that is to be added (deleted).

Figure 5 shows the following execution steps in adding a communication rule. After the new communication rule is stored in the WM, the system deletes those
Figure 5: Execution steps in the incremental validation analysis: adding a communication rule.

associated errors that should disappear after adding this communication rule. For example, an unspecified reception in a communication entity should be deleted if the new communication rule is the missing receive transition; a deadlock should be deleted if the new communication rule is a send transition that can be executed in the deadlock state. The following production rule describes the deletion of a reception error.

\[
\begin{align*}
(p \text{ Add \_delete \_reception \_error} \\
(reception \_error \sim id <s>)
\end{align*}
\]
(new <i>)

(rule "id <i> "entity <e> "type r "cstate <c> "message <m> "nstate <n>)

(globalstate "entity <e> "id <s> "size {<z> > 0} "state <c> "queue <m>)

-->

(removed 1))

The deletion of deadlock errors can be described in a similar way. After these steps, by the data-driven property of the OPS5 production system, the new communication rule will be automatically applied to the suitable global states by triggering the send and receive production rules (described in Section 3.2). Those production rules for checking logical errors and logical properties will also be triggered if this added communication rule introduces new logical errors and logical properties.

Figure 6 shows the following execution steps in deleting a communication rule. If this communication rule does exist, it can further be divided into two conditions: (1) this communication rule has not generated any global state yet; i.e., protocol designers' insight tell this communication rule is useless before this communication rule has ever been applied; (2) this communication rule has been applied and has generated some global states. The following two production rules are used to distinguish these two conditions.

1. (p delete_rule_1

   (old <i>)}
Figure 6: Execution steps in the incremental validation analysis: deleting a communication rule.
(rule "id <i> "entity <e> "type <t> "cstate <c> "message <m> "nstate <n>)

(rule "id {<j> <> <i>} "entity <e> "type <t> "cstate <c> "message <m> "nstate <n>)
- (rule-state "ruleid <j>)
- ->
(remove 1 2 3))

2. (p delete_rule_2
(old <i>)

(rule "id <i> "entity <e> "type <t> "cstate <c> "message <m> "nstate <n>)

(rule "id {<j> <> <i>} "entity <e> "type <t> "cstate <c> "message <m> "nstate <n>)

(rule-state "ruleid <j>)
- ->
(make oldrule <j>)
(make task delete_state)
(remove 1 2 3))

In the first condition, there is no side effect after deleting this communication rule.

In the second condition, after deleting this communication rule, all of the global states generated by this communication rule, all descendants of these global states and the associated information should be deleted. To support these actions, the following
information is recorded in the WM:

1. Those states generated by applying a specific communication rule: This information is stored in element rule-state: (rule-state ruleid <i> stateid <s>). This element denotes that state <s> is produced after communication rule <i> is triggered.

2. The hierarchy among states: This information is stored in element family: (family mother <m> child <c>). This element denotes that state <m> is the mother of state <c> in the reachable global state graph. That is, state <c> is produced from state <m> by triggering a transition.

3. The duplicated states: This information is stored in element samestate: (samestate first <f> second <s>). This element denotes that <f> is the first occurrence of this state and <s> is the duplicated one.

There are three situations result from deleting the associated states:

1. The corresponding generated state is a unique state in the reachable global state graph: In this case, the system deletes this state and its descendants by referring to element family iteratively. The following production rule is used for this case:

   (p delete_state_unique
    (task delete_state)
    (oldrule <i>))
2. The corresponding generated state is a duplicated state: In this case, the generated state is a second occurrence of an existing state. Since this state will not have any descendant in the reachable global state graph, the system deletes this duplicated state only. The following production rule is used for this case:

\( (\text{p delete\_state\_not\_first} \) \\
(\text{task delete\_state}) \\
(\text{oldrule } \<i>\) \\
(\text{globalstate } \<\text{entity 1 } \<\text{id } \<s>) \\
(\text{globalstate } \<\text{entity 2 } \<\text{id } \<s>) \\
(\text{rule\_state } \<\text{ruleid } \<i>\<\text{stateid } \<s>) \\
(\text{samestate } \<\text{first } \<f>\<\text{second } \<s>) \\
\rightarrow \\
(\text{make } \<\text{oldstate } \<s>) \)
3. The corresponding generated state is the first occurrence and there are duplicated states in the reachable global state graph: In this case, the system deletes this state and continues to delete its descendants if they exist. The following production rule is used for deleting the first state and making elements for the following tasks:

```
(p delete_state_first
(task delete_state)
(oldrule <i>)
(rule-state "ruleid <i> "stateid <s>)
(globalstate "entity 1 "id <s>)
(globalstate "entity 2 "id <s>)
(samestate "first <s> "second <n>)
- - >
(make "oldstate <s>)
(modify 4 "id <n>)
(modify 5 "id <n>)
(make changesamestate "first <s> "second <n>)
(make task change_same_state)
(make task delete_more)
(remove 3 6))
```
Since this state can also be produced by triggering other communication rules, the system promotes one of the duplicated states from the element *samestate* to the first occurrence and updates the related information: replacing the value of attribute *first* in the elements of *samestate* with the ID of the promoted one. By using the elements *family* and *samestate* to describe the relationship of global states, the promotion process can be described in a way similar to the above production rules.

When the system deletes a state, all of the associated errors and information should also be deleted. For simplicity, we use the production rule that deletes deadlock errors as illustration.

(p Del_delete_deadlock
(task delete_more)
(oldstate <i>)
(deadlock_error -id <i>)
->
(remove 3))

After deleting the first state, the system continues to delete the descendant states (path B in Figure 6). After deleting one path traced from a state generated by applying this communication rule, the system will continue to delete the other paths traced from those states that were also generated by applying this communication rule (path A in Figure 6). Finally, by triggering those production rules that describe logical errors and logical properties, the system will check whether the absence of this communication rule would lead to any new logical error or logical property.
4.5 PDPS: Protocol Design Production System

Protocol Design Production System (PDPS) is currently developed in the Encore Multimax shared memory multiprocessor machine with 12 processors by using the C-based OPS5 production system [46, 47, 60]. PDPS provides a user-friendly interactive design environment and a knowledge base which can answer all possible questions. Many options are available to protocol designers in the beginning. These options include:

1. specifications of the channel bound, initial states and final states for communication entities;

2. input from either an external file or the designer's terminal;

3. print all error messages immediately?

4. print global states step by step?

5. stop when an error is found or after all errors are found?

6. print the error path (from the initial state to the error state) immediately after an error is found?

7. which definition of the reception error (mentioned in Section 4.3) is used?

PDPS works as follows. After the input and the designer's choices are made, PDPS will start to find all possible legal transitions and print out information. A duplicated state is allowed to be generated, but it will be detected and deleted later.
When some logical errors or logical properties in a certain state are detected, the system can identify these errors or properties and then print out the related messages to the designer. Furthermore, during the intermediate stage of validation, many options are provided, including listing communication rules, currently generated states, and adding/deleting rules dynamically. Since incremental validation is supported in PDPS, only the associated states will be generated or deleted instead of regenerating all states from the beginning. Finally, after generating all possible states either from the beginning or from the point of incremental validation, the system can answer questions, including listing states, communication rules, duplicated states, errors and properties, the states generated by a specified communication rule and the paths from the initial state to a specified state. Meanwhile, the designer can add/delete rules at this point, then the system will continue to do the incremental validation until the designer is satisfied. Appendix Two shows some execution examples in PDPS.

By using the C-based OPS5 executed in the Encore Multimax shared memory multiprocessor machine with 12 processors, PDPS needs 106 seconds to validate the X.25 protocol depicted in Figure 2 [76]. This time includes the system time and the user time. Tables 1 to 12 show the time-queue tables for different process numbers. Table 13 shows the optimal time of each process number and the corresponding queue number for validating the X.25 protocol in PDPS. In addition to specifying the number of processes, the protocol designer can also specify the number of task queues. A task queue holds a list of tasks that are waiting for processing. A task is an independently schedulable unit of pattern match work that can be executed in
Table 1: The time-queue table of validating the X.25 protocol in PDPS when process number equals 1

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>479</td>
<td>483</td>
<td>487</td>
<td>494</td>
<td>502</td>
</tr>
</tbody>
</table>

Table 2: The time-queue table of validating the X.25 protocol in PDPS when process number equals 2

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>476</td>
<td>468</td>
<td>470</td>
<td>478</td>
<td>482</td>
</tr>
</tbody>
</table>

parallel with other tasks. In this way, each process searches for a task by scanning these task queues. A small number of task queues will result in processes' contention for the task queues; on the other hand, a large number of task queues will lead to a situation in which most of the task queues will be empty and processes will waste time scanning several empty task queues before finding one with a task. This table also lists the number of task queues for each number of processes that will produce optimal performance.

Since there are 12 processors and 1 processor is used for processes management, the speed-up reaches its peak when there are 11 processes. When the number of processes is more than 11 (12, 13, 14 ...) the performance becomes worse. The reason is that the overhead for process allocation exceeds the speed-up derived from
Table 3: The time-queue table of validating the X.25 protocol in PDPS when process number equals 3

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>267</td>
<td>255</td>
<td>253</td>
<td>258</td>
<td>266</td>
</tr>
</tbody>
</table>

Table 4: The time-queue table of validating the X.25 protocol in PDPS when process number equals 4

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>199</td>
<td>190</td>
<td>188</td>
<td>189</td>
<td>199</td>
</tr>
</tbody>
</table>

Table 5: The time-queue table of validating the X.25 protocol in PDPS when process number equals 5

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>165</td>
<td>155</td>
<td>152</td>
<td>154</td>
<td>161</td>
</tr>
</tbody>
</table>

Table 6: The time-queue table of validating the X.25 protocol in PDPS when process number equals 6

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>142</td>
<td>139</td>
<td>136</td>
<td>135</td>
<td>139</td>
</tr>
</tbody>
</table>
Table 7: The time-queue table of validating the X.25 protocol in PDPS when process number equals 7

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>135</td>
<td>127</td>
<td>123</td>
<td>122</td>
<td>123</td>
</tr>
</tbody>
</table>

Table 8: The time-queue table of validating the X.25 protocol in PDPS when process number equals 8

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>125</td>
<td>124</td>
<td>120</td>
<td>120</td>
<td>118</td>
</tr>
</tbody>
</table>

Table 9: The time-queue table of validating the X.25 protocol in PDPS when process number equals 9

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>133</td>
<td>121</td>
<td>114</td>
<td>113</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 10: The time-queue table of validating the X.25 protocol in PDPS when process number equals 10

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>127</td>
<td>118</td>
<td>112</td>
<td>109</td>
<td>107</td>
</tr>
</tbody>
</table>
Table 11: The time-queue table of validating the X.25 protocol in PDPS when process number equals 11

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>121</td>
<td>114</td>
<td>110</td>
<td>108</td>
<td>106</td>
</tr>
</tbody>
</table>

Table 12: The time-queue table of validating the X.25 protocol in PDPS when process number equals 12

<table>
<thead>
<tr>
<th>TASK QUEUE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME (second)</td>
<td>138</td>
<td>125</td>
<td>115</td>
<td>109</td>
<td>107</td>
</tr>
</tbody>
</table>

Table 13: The optimal time-process table of validating the X.25 protocol in PDPS

<table>
<thead>
<tr>
<th>PROCESS</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>TASK QUEUE</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>TIME (second)</td>
<td>479</td>
<td>468</td>
<td>253</td>
<td>188</td>
<td>152</td>
<td>135</td>
</tr>
<tr>
<td>SPEED-UP</td>
<td>1</td>
<td>1.024</td>
<td>1.893</td>
<td>2.548</td>
<td>3.151</td>
<td>3.548</td>
</tr>
<tr>
<td>PROCESS</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>TASK QUEUE</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>TIME (second)</td>
<td>122</td>
<td>118</td>
<td>112</td>
<td>107</td>
<td>106</td>
<td>107</td>
</tr>
</tbody>
</table>
multiple processes. A higher degree of parallelism can be explored, because the speed-up obtained from the parallel execution of production rules will multiply with the speed-up obtained from the parallel pattern matching [49, 56, 57, 58]. What allows a high degree of parallel execution of production rules is the fact that the action part of one production rule will not update the condition parts of the other production rules. (Action MAKE makes nonexistent conditions false; actions MODIFY and REMOVE make existing conditions false.) The mutual exclusion issue in parallel protocol validation is many-READ-one-WRITE: after a global state is produced, this global state will be used for (sharable) READ only. The semantics is that after a global state has been recorded in the database, the following access of this global state is query only. Hence, the RHSs (action parts) of production rules for logical errors and logical properties and for the modeling of transitions between global states are independent from one another's LHS. Therefore, a much better performance can be achieved when the parallelism is extended to the parallel execution of production rules.

4.6 Discussion

Formal Description Techniques (FDTs) used for communication protocols can be classified into three categories [73]: the state-transition model [12], the abstract language model [75] and the hybrid model [130, 131]. The state transition model is abstract and is very easy to automate; however, global state reachability analysis is hindered by the problem of state explosion. The abstract language model is good for theoretical proof of functional correctness; however, efforts to prove the correctness of a
program far exceed those required for developing the program, and the correctness proof of a program usually depends heavily on human's ingenuity and is hard to automate. The hybrid model tries to combine the features of both state-transition and abstract language models. Two famous examples of the hybrid model are ESTELLE [130] and LOTOS [131]. In the hybrid model, the state-transition part of the model captures the control aspects of a protocol while variables and data are easily handled by the program part of the model. Depending on how high a level of abstraction is used, those FDTs based on the hybrid model require different degrees of effort in the machine executable phase, either for simulation or real implementation. The OPS5 production system approach is also based on the hybrid model. The OPS5 production system uses production rules to specify the state-transition part of a protocol and uses attributes in elements as generic data templates to specify variables and data of the protocol. Both global state reachability analysis and rule-based program proof techniques [31, 32, 90, 91] for protocol validation are also applicable to the OPS5 production system approach, depending on the complexity of the protocol and the execution environment.

The main advantage of global state reachability analysis is that it is straightforward, well-suited for validating protocols in the state transition model and easy to automate [73, 86, 105]. While global state reachability analysis has been used for validating protocols of low to middle complexity, the practical use of global state reachability analysis for more complex protocols has been hindered by the problem of state space explosion [51, 66, 122]. The reason is that global state reachability analy-
sis has exponential complexity, which leads to state space explosion. Therefore, even with multiprocessor computing systems, it is impractical to use the straightforward reachability analysis to validate every kind of protocol. The reason is that a finite size of storage space and a finite number of processors cannot accommodate and handle an astronomical or infinite amount of information. However, by using multiprocessor computing systems (such as the Encore Multimax with 12 processors and 64Mb memory space in PDPS, or the Hypercube with 128 processors and 128*64 Mb memory space in [37]) that have more powerful computing capability and larger storage space, protocol engineers may be able to use global state reachability analysis for validating more complex protocols. For very complex protocols with an astronomical or infinite amount of global states that are not suitable for global state reachability analysis, program proof techniques for rule-based languages can be used [31, 32, 90, 91]. However, program proof techniques are not so straightforward to apply and not so easy to automate [68]. The difficulty is that the formulation of assertions and proofs often requires human's insight [73, 86, 105].

In the past, many people have tried to reduce the size of global-state space that must be explored [25, 27, 40, 42, 50, 62, 116, 121, 129], but few have tried to use parallel execution for global state reachability analysis [37, 123, 124]. We believe that both the parallel execution and the global state reduction strategy should be applied to obtain the best performance. Moreover, it is not wise to apply one global state reduction strategy to the global state reachability analysis of all kinds of protocols, due to the fact that each kind of protocol has its own characteristics and each global
state reduction strategy has its own applicable domain of protocols. Nevertheless, when a combination of parallel execution and a global state reduction strategy is used, the global state reduction strategy should keep the many-READ-one-WRITE characteristic in protocol validation (discussed in Section 4.5). If the additional information used for the reduction strategy results in many-READ-many-WRITE, then the reduction strategy may not be suitable for parallel execution.

From the experience in constructing a protocol design system based on the OPS5 production system, two disadvantages have been observed. First, although the combined use of the C-based implementation and an efficient pattern match algorithm in the OPS5 production system can resolve the pattern match bottleneck of low to middle complex computations, pattern matching is still the bottleneck for high complex computations in the OPS5 production system. Therefore, for those computations with high complexity, such as global state reachability analysis, special hardware (such as the Encore Multimax with 12 processors) is required. Second, the OPS5 production system is a general-purpose specification language that is not designed for formal protocol specification only. Therefore, some parts of the formal specification cannot be described compactly. For example, $n^2 - n$ elements are required to describe a global state in an n-entity protocol.
CHAPTER V

PROTOCOL IMPLEMENTATION USING PRODUCTION SYSTEMS

This chapter is organized in three sections. Protocol implementation is briefly introduced in Section 5.1. The general implementation model using the OPS5 production system is described in Section 5.2. A brief discussion on the implementation capability of the OPS5 production system is presented in Section 5.3.

5.1 Protocol Implementation

Protocol specifications define those aspects of the protocol entities' behavior which are required for compatible communication among the different communicating systems and required for the provision of the defined communication service. Additional aspects of the protocol behavior are usually left unspecified in the protocol specification; because these additional aspects need to be realized differently in each system implementing the protocol. Therefore, an abstract machine-independent specification of a communication protocol describes the interactions of communication entities; the real implementation of a communication protocol that includes interactions with the execution environment is machine-dependent. Most often, a generic description of the communication protocol is specified in the specification phase by the protocol
designer. In this generic description, the machine-dependent part is usually specified abstractly or left unspecified. The realization of the machine-dependent part, pertaining to the details of the operational architecture and the host operating system, is deferred until the implementation phase. During the implementation phase, the machine-dependent part is coded by the protocol implementer.

Formal specification of communication protocols acts as an interface between the specification phase and the implementation phase. The use of formal specification languages makes it possible to partially automate protocol implementation. In order to enhance the understandability and expressibility and lower the software complexity for the protocol designers, Formal Description Techniques (FDTs) always provide high abstract formulae to specify communication protocols. To achieve automatic implementation, a translator is used between the abstract specification form and the real implementation code [5, 6, 7, 9, 106, 114]. However, if FDTs become too abstract to be used directly in the implementation phase [79, 110], the productivity of automatic protocol implementation is decreased. On the other hand, if protocol designers specify communication protocols using some programming languages [39, 109] to increase the executability of FDTs, then the understandability is decreased and the software complexity is increased.

Based on a globally shared dataspace (working memory) in which different types and levels of information are all represented in a uniform structure (element), the OPS5 production system integrates both rule-based computation formalism and procedure-based computation formalism [14]. Using the OPS5 production system in the speci-
fication phase, the machine-independent part is specified in production rules and the machine-dependent part is abstractly described through external procedure calls with interface elements (abstract primitives). In this way, the protocol designer can use the WM as the communication medium for simulating the interactions of the communicating entities. That is, entity \( x \) inserts the element that represents the corresponding communication message into the WM; similarly, peer entity \( y \) extracts the element that represents the corresponding communication message from the WM. As a result, the simulated execution of communication protocols, either for validating logical correctness or performance analysis, can be achieved without dealing with the real implementation details of the machine-dependent part. Figure 7-(a) shows the abstract architecture for the simulated protocol execution. Using the OPS5 production system in the implementation phase, the details of the machine-dependent part are coded using the procedure-based computation formalism. The inter-communication of the machine-independent and the machine-dependent parts is achieved by the insertion/extraction of elements that represent the corresponding abstract primitives into/from the WM. The machine-dependent part then invokes the real communication facility to send the messages to the peer entity. That is, the machine-dependent part extracts communication messages from the interface element and then invokes the communication mechanism to really transmit the messages. Similarly, the machine-dependent part inserts the corresponding elements into the WM after receiving messages from the communication mechanism. In this way, two geographically separated rule-based communication systems are generated to achieve protocol entities'
inter-communication. Figure 7-(b) shows the abstract architecture for the protocol execution of the real implementation.

Since computations in the OPS5 production system are based on pattern matching, all of the attributes of elements in production rules or in external procedure calls can act as data templates for generic data types. This capability enhances the generic specification that allows different realizations for various implementation environments.

5.2 General Implementation Model in OPS5

Figure 8 shows the general implementation model for the OPS5 production system approach. Production rules for the modeling of transitions are stored in the production memory. Elements in the WM can be classified into three sets:

1. the specification of communication rules.
2. the representation of abstract service primitives.
3. other elements, such as the elements used for states, variables and predicates.

The above production rules and elements are specified by the protocol designer.

The Inference Engine (IE) acts as the dispatcher to select one of the applicable transitions for execution. The Interface Data Structures (IDS) store the real data formats that are used in communication with other layers. The external procedures (and functions) can be classified into the following three groups:
Figure 7: Abstract architectures for protocol execution: (a) the simulated case; (b) the real implementation case.
Figure 8: Implementation Model
1. Input Interface Procedures (IIP): IIP includes the upper-layer input interface procedures and the lower-layer input interface procedures. In the first step, IIP decodes the input events received by the incoming event process according to IDS. Then, IIP encodes the incoming events in the elements that represent the corresponding abstract service primitives. By using the commands provided in OPS5, these interface elements can be inserted into the WM. For example, the following commands insert element *DataIndication*, which is used for data indication in X.25, into the WM:

```plaintext
dollar_value(dollar_intern("DataIndication"));
dollar_tab(dollar_intern("entity"));
dollar_value(dollar_cvna(EID));
dollar_tab(dollar_intern("message"));
dollar_value(dollar_intern(userbuffer[EID]));
dollar_assert();
```

where variables *EID* and *userbuffer[EID]* represent the identifiers of the receive entity and the pointer pointing to the storage of the receive data block, respectively. Since every data item in OPS5 is regarded as an atom, function *dollar_intern* is used to translate a string into a string atom and function *dollar_cvna* is used to translate a number into a number atom. After the translation, function *dollar_value* puts the assigned value into the current field indicated by a pointer implicitly. In order to refer to a specific attribute, function *dollar_tab* is provided to move the pointer to the attribute expressed in the
argument. Finally, when all values are assigned, function `dollar_assert` inserts element `DataIndication` into the WM.

2. Output Interface Procedures (OIP): OIP includes the upper-layer output interface procedures and the lower-layer output interface procedures. In the first step, OIP decodes the abstract primitives that are represented in some interface elements. Then, OIP encodes the corresponding outgoing events according to IDS. By using the commands provided in OPS5, OIP decodes the abstract primitives by extracting the values from the interface elements. For example, the following commands extract the values of attributes in element `DataRequest` used for data transmission in X.25:

\[
\begin{align*}
EID &= \text{dollar_cvan}(	ext{dollar_parameter}(	ext{dollar_litbind}(	ext{dollar_intern}("entity")))); \\
\text{buffer}[EID] &= \text{dollar_cvas}(	ext{dollar_parameter}(	ext{dollar_litbind}(	ext{dollar_intern}("message"))))
\end{align*}
\]

where variable `buffer[EID]` records the pointer pointing to the storage of the messages to be transmitted. In order to obtain the value of a specific attribute, the index of the attribute should be known first. Function `dollar_litbind` is provided to return the index of the field of the attribute indicated by the argument. Then, function `dollar_parameter` returns the atom of the field indicated by the argument. Thereafter, to obtain the value of the atom with the right data type, function `dollar_cvan` is used to translate a number atom into a number; function `dollar_cvas` is used to translate a string atom into a string.
3. Local Event Procedures (LEP): LEP includes procedures and functions that deal with the local system events, such as memory management and the time-out monitor.

The control flow in this model (Figure 8) is described as follows:

1. When an input event arrives from the other layers, the corresponding incoming event process is awakened and the corresponding input interface procedure is called.

2. Next, this input interface procedure interprets the input event according to IDS and generates the corresponding elements.

3. Then, the generated elements are inserted into the WM. These elements, in turn, trigger the corresponding production rules by IE and invoke the related actions: changing a local entity's state, modifying the configuration of the WM and/or calling external procedures.

4. After that, these called external procedures may invoke some outgoing events by extracting the values from the interface elements. According to these extracted values and IDS, the output interface procedure generates the outgoing message and/or some local environment events. For example, a timer is set up to monitor a transmission event.

5. Finally, the outgoing event process sends the outgoing message to the other layers.
For local event processing, such as the initiation and the expiration of a timer, the corresponding execution is similar to that for an outgoing event process and for an incoming event process, respectively.

5.3 Discussion

For both simulation, either for validating logical correctness or performance analysis, and implementation, there is a trade-off between the abstract specification level and the complexity in deriving the machine executable code. As has been pointed out in [79, 110], LOTOS specifications often tend to be too abstract so that they cannot be directly translated into the machine executable code; therefore, they need a series of translation steps, such as LOTOS specifications $\rightarrow$ CCS specifications $\rightarrow$ PROLOG specifications [77], toward the executable validation level; or they need human's guidance and intervention in an interactive compiler to translate LOTOS code into C code [79, 110]. On the other hand, if a Formal Description Technique (FDT) becomes an overspecification in order to be directly machine executable, it is no longer qualified as an abstraction. As has been pointed out in [30, 107, 113], ESTELLE specifications suffer from this kind of drawback, since they include too many implementation-oriented details based on the extended PASCAL notation. Therefore, one of the motivations using the OPS5 production system approach is trying to compromise these two endpoints: the OPS5 production system provides both rule-based and procedure-based computations to avoid overspecifications, and the OPS5 production system provides generic data templates by using pattern matching for generic realizations.
In the OPS5 production system approach, the machine-independent part is specified in rule-based computation formalism and the machine-dependent part is abstractly specified through external procedure calls by the protocol designer in the specification phase. In the implementation phase, the machine-dependent part is coded according to the operational architecture and the host operating system by the protocol implementer. The flexibility of the OPS5 production system also allows the implementer to code the machine-dependent part using production rules, depending on whether the computation is more production-oriented or procedure-oriented. For example, the control arbitration for priority-based events can be expressed in production rules (discussed in Section 3.2).

For the code process paradigm, the formal specification in the OPS5 production rule-based code is translated into assembly code by the ParaOPS5 compiler, then the assembler translates assembly code into object code. This object code can be executed on Encore Multimax, VAX workstations, VAX uniprocessor machines (such as VAX 11/780), or VAX multiprocessor machines (such as 8800 and 11/784) [60]; for the external procedures or functions that are written in a procedure language, such as C or PASCAL, the corresponding compiler compiles them into object code. Next, by using the linker and the loader, the complete object code for the communication protocol is produced. In order to allow the OPS5 code to be directly executed on a general UNIX system, instead of translating to assembly code and then object code, a CParaOPS5 compiler can translate the OPS5 production rule-based code into C code (which can be executed in either uniprocessor systems or multiprocessor systems).
this way, the OPS5 specification is allowed for direct execution on a general UNIX system.
This chapter is organized in six sections. The rationale of the incremental test method is described in Section 6.1. The traditional test sequence generation techniques are introduced in Section 6.2. The traditional test architectures are briefly presented in Section 6.3. Then, the incremental test sequence generation method and its formal modeling in OPS5 production rules are explained in Section 6.4. The prototypes of the incremental test architectures are described in Section 6.5. Finally, a brief discussion on the incremental test method is given in Section 6.6.

6.1 Protocol Testing

To ensure that the implementation of a communication protocol conforms to the original specification, a protocol test process is invoked to check for conformance. In the traditional approaches, heterogeneous representation mechanisms are used in the design phase and the test phase. Therefore, the Implementation Under Testing (IUT) is regarded as a black box with two ports: an input port and an output port. IUT is tested for conformance by applying a number of test sequences. These test sequences consist of sequences of input messages and output messages that are derived from the
formal protocol specification [73]. When the protocol testing is invoked, the external testers (test engine) apply input messages to IUT and verify whether output messages match with those expected in the test sequences. If they are matched, IUT is said to conform to the specification; otherwise, IUT is faulty [104].

Based on the formal state transition specification model, the key issue in protocol testing is that every transition in the original specification should be tested at least once [82, 95]. Under the traditional concept of “a black box with two ports,” due to the lack of observability, each state is associated with a distinguishing sequence [24, 38, 92]. Therefore, one can verify a state by observing the output message sequence after the input message sequence of this state’s distinguishing sequence is applied. Due to the lack of controllability, a sequence of transitions should be applied to reach the head state of the transition to be tested. Therefore, a reset transition from each state to the initial state should be added to the specification and implementation to control the test process; or a sequence of transitions is executed to switch from the tested transition to the transition to be tested. As a result, the testing of a transition is divided into three steps:

1. First, a sequence of transitions is applied in order to reach the head state of the transition to be tested.
2. Next, the input/output messages of the transition under testing are applied.
3. Then, the corresponding distinguishing sequence of the entrance state is applied.

Consequently, if the transition, such as time-out or connection establishment, that
needs a long time to be executed cannot be avoided, it will take a long time to perform the testing [70, 94]. Moreover, most of the execution time for protocol testing will be wasted in executing the following transitions:

1. the sequences of transitions used to switch to the head states of the transitions to be tested.

2. the sequences of transitions used to verify the entrance states of the transitions under testing.

Even some optimization techniques are applied to minimize test sequences [2, 18, 21, 22, 28, 80, 97, 112, 117], some transitions still need to be executed repeatedly. The repeated execution of the above transitions makes protocol testing very time consuming [87, 94].

When a Formal Description Technique (FDT) is used to formally specify the protocol and the corresponding translator of the FDT can generate the complete machine executable code of the protocol implementation, then protocol conformance testing is no longer required if the translator can guarantee to generate error-free machine executable code. Unfortunately, the protocol implementation consists of two parts, the machine-independent part and the machine-dependent part. Although the machine-independent part can be mechanically translated into the machine executable code, the machine-dependent part should be manually encoded by protocol implementors. The reason is that the realization of the machine-dependent part relies on the operational environment of the communication protocol and is therefore machine-dependent. In this way, after the integration of the machine-independent
part and the machine-dependent part, the test process still needs to be invoked to check the conformance. Nevertheless, when new and high level formal representation mechanisms, such as the existing FDTs, are used for formally modeling communication protocols, one problem should be considered: Do we need to follow the old principles that are based on the original (low level) representation mechanisms for protocol design processes (such as the conformance testing)? Conceptually, the effect of the high level representation in software is similar to the effect of transforming massive circuits (consisting of transistors, resistors, capacitors, et al.) to some chips in hardware. When circuit design engineers use some chips to achieve some functions, they only need to understand the functionality of each pin in the chips and then connect the related pins together without having to know the details of the circuits inside the chips. The same concept could be applied to software. Especially, when the software is represented formally, abstractly and modularly, and the software size can therefore be reduced dramatically, e.g., from \( n \times \{10000, 1000\} \) lines to \( n \times \{1000, 100\} \) lines. Therefore, protocol engineers could consider replacing “a black box with two ports” with “software chips with multiple ports” when high level formal representation mechanisms are used.

Although the existing formal specification of a communication protocol is executable and therefore the test sequences can be mechanically derived using some computer-aided methods, protocol conformance testing still utilizes the concept of regarding IUT as “a black box with two ports.” To simplify the protocol testing, rather than proposing some optimization algorithms to minimize the test sequences
In the existing research, an incremental protocol test method that is based on modifying test architectures and enhancing FD'Ts' functionalities so that they are powerful enough to be used in specification, implementation, and test phases is proposed in this dissertation. The key idea of the incremental protocol test method is to add a module that provides the capability of direct control and observation of IUT in the local side. Since the formal protocol specification part is executable and is regarded as some software chips with multiple ports, test sequences can be mechanically generated directly based on the formal protocol specification part of IUT [55].

To have direct control and observation,

1. a single high level representation mechanism, which is currently the OPS5 production system, is used in both the design phase and the test phase;

2. a Test sequence Generator and state Monitor (TGM) is added in the local environment.

By using a single representation mechanism approach, the process of protocol testing consists of two parts:

1. The specification of test sequence generation.

2. The inter-communication between this specification module and external testers.

Since the first part is based on the machine-independent specification of a communication protocol, the specification of test sequence generation is environment-independent. Since the second part is based on the host environment of IUT and external testers, the realization of inter-communication between the module of test
sequence generation and external testers is environment-dependent. Using a globally shared dataspace (working memory) in which different types and levels of information are all represented in a uniform structure (element), protocol test sequence generation can be formally represented in production rules by combining those related elements in the formal protocol specification with additional elements used for controllability and observability in the incremental test process. The inter-communication between the module of test sequence generation and external testers is specified abstractly and its realization in procedure-based computation formalism relies on the host environment of IUT and external testers. Conceptually, the combination of related elements between IUT and TGM is similar to the composition of hardware chips that is achieved by plugging in components and connecting pins. Figure 9 shows the abstract test architectures of the traditional test approaches and the incremental test approach, where the lower tester can be located either locally or remotely.

With the improved controllability, the status of IUT can be set to the head state's status of the transition to be tested; with the improved observability, the entrance state of IUT can be directly verified by checking the IUT's state recorder after the application of a transition. As a result, by keeping track of the status of tested events, the incremental test method eliminates the possibility of repeated execution of some transitions in the communication protocol under testing. In other words, through the use of a single representation mechanism, the translators used to translate the formal protocol specification into the derivation of protocol test sequence are eliminated; through the globally shared dataspace, the complex procedures used
Figure 9: The abstract test architectures: (a) the traditional test approaches; (b) the incremental test approach.
to find distinguishing sequences of all states, the complex procedures used to find (optimal) switching sequences between tested transitions, and the execution of the time-consuming input/output sequences used to switch the states and to verify the entrance states of IUT can be eliminated.

6.2 Traditional Test Sequence Generation Techniques

The protocol behavior that specifies how to handle reception of messages, time events, etc., is referred to as core behavior [92]. Using the deterministic state-transition-based specification, since a real protocol may not cover all possible state-input combinations, it is assumed that the protocol entity produces null outputs and remains in its current state for the unspecified state-input combinations [92]. These self-loop transitions are called non-core behaviors [92]. Therefore, protocol testing is defined at two levels: strong conformance testing and weak conformance testing [92].

1. Strong conformance testing: A protocol implementation has strong conformance to its original specification if all of its input/output sequences are the same as those in the original specification.

2. Weak conformance testing: A protocol implementation has weak conformance to its original specification if it has the same core behaviors as the original specification; however, it has unspecified behaviors for those non-core behaviors.

In the traditional approaches, heterogeneous representation mechanisms are used in specification, implementation, and test phases. Therefore, some translators, such as those used for ESTELLE [93] and for LOTOS [44], are used to translate the formal
protocol specification into the derivation of test sequences. As a result, the test engine lacks the direct controllability and observability for IUT. An input sequence is said to be a distinguishing sequence if the output sequences produced are different for all states. This sequence can be used to ascertain the entrance state because it generates a unique output sequence for each state. A characterization set consists of a set of input sequences, such that the last output messages obtained from the application of these sequences are different for all states. In [38], a distinguishing sequence is generated for a CFSM-based specification, whereas in [24], a characterization set is produced for a CFSM-based specification. However, not every protocol specification can generate a distinguishing sequence for each state by using the above two methods. In [92], a unique input/output (UIO) sequence is derived for each state. The UIO sequence of each state is an I/O behavior that is not exhibited by any other state. Since not every state can have a UIO sequence, a signature is generated for those states that have no UIO for identification. In these three methods, most of the execution time will be wasted in executing the transitions used to switch to the head states of the transitions to be tested and in executing the transitions used to verify the entrance states of the transitions under testing. Figure 10 shows the abstract test procedure of the traditional test approaches.

6.3 Traditional Protocol Test Architectures

To have a protocol testing, protocol engineers need on the one hand a way to generate the test sequences, and on the other hand a mechanism to check actual behavior of an implementation against the original specification. Test event generation consists in
principle of enumerating sequences of allowable and non-allowable interactions. Trace checking consists of ascertaining whether the response of IUT for a particular event is consistent with the specification.

In this section, traditional test mechanisms (architectures) of the local method, the distributed method, the coordinated method, the remote method [70, 71, 89, 94] and the ferry control method [125, 126, 127, 128] are briefly introduced.

In the traditional approaches, test architectures usually consist of three parts: the Upper Tester (UT), the Implementation Under Testing (IUT), and the Lower Tester (LT) that is located either locally or remotely (Test Management Center is regarded as part of LT). LT is the mechanism for providing control and observation of events at IUT's bottom interface. UT is the mechanism for providing control and observation of
events at IUT's upper interface. LT and UT stimulate IUT by exchanging test events at the bottom and top interfaces of IUT. In addition, there are Test Coordination Procedures (TCPs) conceptually that provide means of cooperation between LT and UT during the protocol testing.

The local method uses control and observation of the abstract service primitives (ASPs) defined at the service boundaries directly above and below IUT. That is, LT controls and observes the (N-1)-ASPs and (N)-PDUs, while UT controls and observes the (N)-ASPs. TCPs are defined in terms of the ASPs of the protocol under testing. Figure 11 shows the abstract test architecture of the local method.

When the client of IUT wants to simulate a more realistic communication testing and therefore a physical networking media is used, LT and UT are geographically separated. In this way, LT and UT are connected by an underlying OSI service that provides the (N-1)-service. The distributed method uses control and observation of (N)-PDUs and the (N-1)-ASP's defined at the service boundary between LT and (N-1)-service provider, and control and observation of the (N)-ASPs defined at the service boundary between IUT and UT. TCPs are also defined in terms of the ASPs of the protocol under testing. Moreover, the distributed method relies on the protocol under testing to resolve synchronization between UT and LT. Figure 12 shows the abstract test architecture of the distributed method.

The coordinated method is an enhanced version of the distributed method. The coordinated method differs from the distributed method in the following two features:

1. A standardized Test Management Protocol (TMP) is used for TCPs;
Figure 11: The abstract test architecture of the local method
Figure 12: The abstract test architecture of the distributed method
2. No exposed upper interface between IUT and UT.

Therefore, the synchronization between UT and LT is resolved through TMP in the coordinated method. Figure 13 shows the abstract test architecture of the coordinated method.

The remote method assumes no interface at the top of IUT and no explicit TCP (if the coordination is needed, it is manually accomplished). The remote method uses control and observation of (N)-PDUs and the (N-1)-ASP's defined at the service boundary between LT and (N-1)-service provider. Figure 14 shows the abstract test architecture of the remote method.

The ferry control method is a modification of both the distributed and coordinated methods. In the ferry control method, UT is moved from the client environment to the remote test laboratory. This method simplifies the synchronization problem between UT and LT, and simplifies the software needed in the client side. A ferry control protocol (FCP) is used to provide services between IUT and LT and UT. Figure 15 shows the abstract test architecture of the ferry control method.

6.4 Formal Specification of the Incremental Test Sequence Generation

In order to avoid traversing some transitions repeatedly, an incremental test procedure can be used to ensure that each transition is tested exactly once [55]. This method is based on the following features:

1. State monitoring: When a transition has been tested, the entrance state of IUT is verified directly instead of using distinguished sequences. Using the WM, the
Figure 13: The abstract test architecture of the coordinated method
Figure 14: The abstract test architecture of the remote method
Figure 15: The abstract test architecture of the ferry control method
state monitoring can be achieved by applying production rules that keep track of the state of IUT.

2. Status backtracking: When all of the transition paths that originate from a state have been explored, the test sequence is searched backward to an ancestor state from which some unexplored transitions originate and then IUT's status is rollbacked to this ancestor state's status. Using the WM, the status backtracking can be achieved by applying production rules that keep track of the test history (test sequences).

In our approach, the procedure for generating test sequence is based on a depth-first search: when a transition with input/output I/O from state $S_i$ to state $S_j$ is tested, the next transition to be tested is from $S_j$ to $S_k$, if such a transition exists; otherwise, the test path is backtracked to state $S_i$ or $S_i$'s ancestor state from which some untested transitions originate. Figure 16 shows the abstract test procedure of the incremental test approach.

Since the rule-based protocol specification is the partial code of the real implementation and the test sequence generation would be more simply represented by using the nonexistence condition of communication rules, elements rules that represent communication rules are duplicated into elements transitions to have a simplified modeling of the test sequence generation. In this way, whenever a communication rule (state transition) has been tested, the corresponding transition is deleted. The following production rules are used to generate the pseudo communication rules transitions for entity DTE (the test sequence generation for DCE can be represented similarly).
reset to initial state

apply input $I(i-j)$ to transition $T(i-j)$ and verify output $O(i-j)$

status backtrack

verify state $S(j)$

Figure 16: The abstract test procedure of the incremental test approach

1. (p transition_edge_receive
   (rule "entity DTE "type r "estate <cs> "message <m> "nstate <ns>)
   - (transition "in_from DCE "in_message <m> "current_state <cs>)
   - - >
   (make transition "in_from DCE "in_message <m> "current_state <cs> "next_state <ns>)

2. (p transition_edge_send
   (rule "entity DTE "type s "estate <cs> "message <m> "nstate <ns>)
   - (transition "out_to DCE "out_message <m> "current_state <cs>)
   - - >
   (make transition "out_to DCE "out_message <m> "current_state <cs> "next_state
Figure 17 shows the execution procedure of the incremental test sequence generation. The formal representation of the incremental test sequence generation specified in OPS5 production rules is described as follows:

1. Set the current state $S_c$ and the entrance state $S_e$ of IUT as the initial state $S_i$.
   This can be represented in the following production rule:

   $$(p \text{ Sequence Generation Initial}
   (\text{initial state } \sim \text{state } <is>))
   (\text{Inform } \sim \text{state } <cs>)
   - ->
   (\text{modify 2 } \sim \text{state } <is>)
   (\text{make enter state } \sim \text{state } <is>))$$

   where element $\text{initial state}$ records the initial state of IUT; element $\text{enter state}$ records the entrance state of IUT after the application of the transition to be tested;

2. Nondeterministically choose an unexplored transition $S_c \xrightarrow{I/O} S_n$. Input $I$ is applied and the output is checked to verify whether it is $O$. This can be represented by the following production rule:

   $$(p \text{ Sequence Generation}
   (\text{Inform } \sim \text{state } <cs>))$$
Figure 17: The execution procedure of the incremental test sequence generation
(enter_state ^state <es>)
{(state_stack ^s stk <st>) <st stk>}
(transition ^in_from <if> ^in_message <im> ^out_to <ot> ^out_message
<om> ^current_state <cs> ^next_state <ns>)

->
(modify 2 ^state <ns>)
(modify 3 ^s stk <cs> (substr <st stk> 2 inf))
(call event_generation event ^in_from <if> ^in_message <im> ^out_to
<ot> ^out_message <om>)
(remove 4))

where element state_stack records the order of traversed states. In order to have the depth-first process, the first-in-last-out stack-like access strategy is used to store the traversed states in the second action. Element event contains I/O messages and I/O interaction points of the transition to be tested. External procedure event_generation with element event will activate UT or LT to create the corresponding input message to trigger IUT and to record the output message from IUT.

3. Verify whether the entrance state of IUT is correct or not. When IUT has an unexpected entrance state, IUT is erroneous. The state verification is realized by the following production rules:
(p Entrance_State_Matched
(Inform "state <s>)
(enter_state "state <s>)
- ->
(call state_report state_verification "result matched))

(p Entrance_State_Unmatched
(Inform "state <cs>)
(enter_state "state {<es> <> <cs>})
- ->
(call state_report state_verification "result unmatched)

where external procedure state_report with element state_verification will activate the test management center to log the result.

4. If there is an unexplored transition originating from state $S_n$, then keep $S_n$ as the current state and apply the production rule in step 2; otherwise, search backward to find an ancestor state $S_a$ of $S_n$ from which some unexplored transitions originate. The following production rule describes the state backtracking:

(p State_Backtrack
(Inform "state <cs>)
- (transition "current_state <cs>)
{(state_stack "s_stk {<st> <> nil}) <ststk>}

In order to express the condition that the current state is out of unexplored transitions, the same variable is used in the attributes state of element Inform and current_state of element transition. In the RHS, the first action changes the current state of IUT to the ancestor state. The second action pops out the head item from element state_stack. This production rule is repeatedly applied until a state from which some unexplored transitions originate is found. Then the production rule in step 2 is applied to continue the test process.

5. When state $S_a$ from which some unexplored transitions originated does not exist, it implies that all transitions have been explored and this test session can be terminated. The following production rule describe the termination condition.

```
(ps Protocol_Test_Over
  (state_stack ^s stk {<st> = nil})
  ->
  (call event_generation test_over))
```

When element state_stack contains no traversed state, it means that all of the transitions have been tested. External procedure event_generation with element test_over notifies the test management center to terminate the test session.
Test sequences for checking the strong conformance cover core transitions as well as non-core transitions. To have the strong conformance testing, it is required to add those missing transitions to the original specification such that the protocol specification becomes fully specified. These artificial transitions produce null as output and IUT remains in the same state. The following production rules produce these artificial transitions:

1. (p input_message_type
   (rule "entity DTE "type r "message <m>")
   - (message "in_from DCE "message <m>)
   - ->
   (make message "in_from DCE "message <m>))
This production rule generates all of the receive messages with their input points.

2. (p state_type_1
   (rule "entity DTE "cstate <cs>)
   - (statetype "entity DTE "state <cs>)
   - ->
   (make statetype "entity DTE "state <cs>))
(p state_type_2
 (rule "entity DTE "nstate <s>)
 - (statetype "entity DTE "state <s>)
The above two production rules generate all of the states of IUT (for entity DTE). The second production rule is used for terminal states.

3. (p artificial_transition_generation
   (message "in_from DCE "message <m> )
   (statetype "entity DTE "state <cs> )
   - (rule "entity DTE "type r "cstate <cs> "message <m> "nstate <ns> )
   - (transition "in_from DCE "in_message <m> "current_state <cs> )

   ->
   (make rule "entity DTE "type r "cstate <cs> "message <m> "nstate <ns> )
   (make transition "in_from DCE "in_message <m> "current_state <cs> "next_state <cs> ))

   This production rule generates the transitions for those states in which the receptions of some messages are not specified originally.

   For the testing of the X.25 protocol depicted in Figure 2, when IUT is the DTE (DCE) entity, it is assumed that the test engine will generate the expected reactions of the DCE (DTE) entity. That is, whenever the input or output point is DCE (DTE), it is the responsibility of the test engine to send or receive the corresponding messages, either locally or remotely.

   To have a comparison between the traditional test sequence generation methods and the incremental test sequence generation method, the transmitter section of
the half duplex ABP [92] is used as an example in Appendix Three. Furthermore, Appendix Three also shows the formal modeling of the incremental test sequence generation when some variables (in addition to the protocol entity’s state recorder) exist in the formal protocol specification.

6.5 Prototypes of the Incremental Test Architectures

In this section, the prototypes of the incremental protocol test architectures are presented.

In the incremental test architectures, a Test sequence Generator and state Monitor (TGM) is added. TGM provides the functionalities of generating test events and monitoring the state of IUT. In TGM, based on the integration of rule-based and procedure-based computation formalisms in the OPS5 production system [14], the environment-independent test sequence generation is formally represented in rule-based formalism by combining those related elements in the formal protocol specification with additional elements used for observability and controllability in the test process; the environment-dependent inter-communication part is realized by using the procedure-based formalism according to the host environment of IUT and external testers. TGM and IUT communicate with each other through the insertion/extraction of elements into/from the WM. Furthermore, TGM uses external procedure calls to create communication channels with UT and LT. Through these communication channels, TGM will notify UT and LT to send/receive the corresponding input/output messages to/from IUT; UT and LT can inform TGM their readiness for the next test event.
Figure 18 shows the prototype of the incremental test architecture for the local method. In the local method, test coordination procedures (TCP) among UT, LT, TGM, and the Test Management Center are realized using some intra-machine communication channels. These communication channels can be realized using the interprocess communication facilities provided by the operating systems of the host environment of IUT and external testers. All of the test results, either successful or nonsuccessful, are logged in the Test Management Center for future analysis.

Figure 19 and Figure 20 show the prototypes of the incremental test architectures for the distributed method and the coordinated method. Since LT is located remotely, some inter-machine communication channels are required to achieve the coordination procedures between TGM and LT and the Test Management Center. These communication channels need to utilize the services provided by the existing communication networks. That is, communication between TGM and LT and the Test Management Center are achieved using the protocol under testing (the distributed method) or the services provided by the Test Management Protocol (TMP) (the coordinated method), where TMP is implemented in the Test Management Protocol Machine.

Figure 21 shows the prototype of the incremental test architecture for the remote method. Since there is no additional communication channel in the remote method, the communication between TGM and LT and the Test Management Center is achieved manually, for example, using a telephone or a terminal between these two operators [89].

Figure 22 shows the prototype of the incremental test architecture for the ferry
Figure 18: The prototype of the incremental test architecture for the local method
Figure 19: The prototype of the incremental test architecture for the distributed method
Figure 20: The prototype of the incremental test architecture for the coordinated method.
Figure 21: The prototype of the incremental test architecture for the remote method
Figure 22: The prototype of the incremental test architecture for the ferry control method
control method. This method is similar to the distributed method and the coordinated method, except that UT is located remotely. Since a Ferry Control Protocol (FCP) is provided to achieve the communication between IUT and UT and LT, the communication between TGM and UT, LT and the Test Management Center can also be achieved through the services provided by FCP, where FCP is implemented in the Ferry Control Protocol Machine.

The flow control of the incremental test method is as follows:

1. TGM sets IUT's current state according to the event to be tested.

2. TGM generates and sends the information of the selected test event to external testers (using the test coordination protocol for the remote tester).

3. UT, LT or IUT generates the corresponding I/O event (which can be realized automatically by programs or manually by an operator in external testers).

4. Test Management Center logs all of the test results (automatically or manually).

5. UT and LT notify TGM about their readiness for the next test event (using the test coordination protocol for the remote tester).

6. TGM executes status backtracking for IUT, then goes to step 1 if the testing is not over; otherwise TGM ends this test session.

6.6 Discussion

One of the main differences between the incremental test approach and the traditional test approaches is the site of the test sequence generation module. In the incremental
test approach, the test sequence generation module is extracted from the remote site to the local site. That is, TGM is the active test event generator and external testers become passive test event generators that are notified (controlled) by TGM. The other functions are still in external testers. In this way, external testers cooperate with each other for conformance testing. For example, if the current test event is a timeout event, the external tester will wait for a period of time (> the timeout interval), then send the corresponding message (or just do nothing) and observe the response from IUT.

In the traditional test approaches, the synchronization problem may occur in some conditions [23, 72]. For example, there is a synchronization problem in the following two consecutive test events: $I_1/O_1, I_2/O_2$, where $I_1$ is input from LT, $O_1$ is output to LT, and $I_2$ is input from UT. In this example, UT does not know when it should send the message and therefore IUT may receive the message from UT before receiving the message from LT; or the message from LT is delayed in the communication medium and therefore the message from UT is received earlier than the message from LT, even if UT actually sends the message after LT has sent the message. Since TGM is the active test event generator and a new event (a transition to be tested) is generated after the former one is tested, the synchronization problem between UT and LT is resolved correspondingly. The synchronization problem is resolved in TGM by using the test coordination protocol, such as TMP, FCP, or operators. That is, UT or LT can inform TGM after it has sent (received) the corresponding message for IUT's input (output) event. Therefore, when TGM receives the readiness message for the
next event from UT or LT, and when the local execution privilege is switched to TGM after IUT has executed the corresponding input/output events, TGM will initiate the next test event. Consequently, the synchronization problem between two consecutive I/O test events can be resolved.

In addition to applying debugging tests performed by the implementation team, protocol software development projects usually involve the development of test events from the specifications and the application of these test events to the implementation by the test team. In the traditional test approaches, it is controversial who— the designer, the implementer, or the tester—should specify the test events. Therefore, the test of IUT always involves significant coordinating effort between the test laboratory and test clients. Much of the effort is covered in the completion and review of the Protocol Implementation Conformance Statement (PICS) and the Protocol Implementation eXtra Information for Testing (PIXIT), which provide specific information about IUT to the test laboratory [3]. Using the incremental test approach, the test events generation can be formally modeled and then automatically generated in TGM by the test client (the designer and/or the implementor). The formal specification of the automatic generation of test events that is achieved by the client reduces the coordinating effort, either oral coordinating effort or document coordinating effort, between the test laboratory and the test client, because the test laboratory behaves as a mechanically passive test event generator that is notified by TGM. Furthermore, based on the high level representation, the automatic generation of test events embedded in TGM can easily adapt to specification changes.
The generation of test sequences and the check of actual behavior are interleaved according to the algorithm described in Section 6.4. The interleaved approach is suitable for the local method, the distributed method, the coordinated method, and the ferry control method. The incremental test procedure can also be divided into two sequential phases as that in the traditional test approaches: all of the test sequences are generated and stored in element *events* in the first phase; then based on the generated test sequences that are stored in the WM, the second phase is invoked to check the actual behavior of IUT. In this way, additional information is required in element *event*, such as the head state and the tail (entrance) state of the event. As a result, TGM can set IUT to the corresponding state and verify whether the entrance state of IUT is correct or not according to the information stored in element *event*. The two-phase approach is suitable for the remote method, because the interleaved approach will result in too much manual communication. In the two-phase approach for the remote method, a duplicated copy of the test events can be batched to the remote site after the first phase is over. Additionally, an identifier can be assigned to each event to simplify the notification of which event is going to be tested. The two-phase approach is similar to the traditional test approaches, except that the test events are actively generated in the remote site in the traditional test approaches.

Compared with the traditional test architectures, the main drawback of the incremental test architectures is the addition of software (developed in TGM) in the client environment. However, the effort spent in the development of the software in TGM is reduced by the utilization of the high level formal representation mechanism.
Furthermore, the additional work load for TGM is compensated for by eliminating the complex procedures used to find distinguishing sequences of all states and the complex procedures used to find (optimal) switching sequences between tested transitions, and is also compensated for by saving the time spent in the execution of the time-consuming input/output sequences used to switch the states and to verify the entrance states of IUT.
CHAPTER VII

CONCLUSION

This chapter first summarizes the main research results in this dissertation. Then, conclusions and future research issues are given.

7.1 Summary

The major contribution of this research is to propose a computer-aided protocol design methodology based on a single representation mechanism, the OPS5 production system, to formally model all of the four phases in communication protocol design. Using the single representation mechanism approach, a Protocol Design Production System (PDPS) has been developed in the Encore Multimax multiprocessor machine to facilitate protocol design. In PDPS, a user-friendly interactive environment is supported with an incremental validation process and a knowledge base is provided to answer all possible questions. Since control is separated from knowledge in PDPS, protocol designers obtain the advantages of clear knowledge representation, strong modularity, modifiability, and expressibility. The incremental validation process provided in PDPS shows its effectiveness and usefulness when the modification is small. Furthermore, based on the use of a single representation mechanism for protocol design and based on modifying test architectures, an incremental protocol test method
is also proposed. By keeping track of the status of tested events, the incremental test process eliminates the possibility of repeated execution of some transitions in the communication protocol under testing. Figure 23 shows an overview of the protocol design process using the OPS5 production system approach.

Chapter II of this dissertation introduces the OPS5 production system. In order to have a general view, the architecture, notion, and the conflict resolution strategy are briefly presented. The external procedure call, which is one of the main considerations for the proposed methodology, is also discussed. Finally, the RETE algorithm, which is used in the OPS5 production system to resolve the pattern match bottleneck, is given.

Chapter III is devoted to protocol specification using the OPS5 production system. A brief introduction to formal protocol specification, including the motivation, key issues, and traditional approaches, is given. The formal modeling of communication protocols using the OPS5 production system is explained and exemplified. Referring to Figure 23, elements that represent communication rules (state transitions), abstract service primitives, predicates and variables, and production rules for the modeling of transitions are formally specified by the protocol designer. The OPS5 rule-based representation allows protocol designers to specify protocols either very abstractly or in detail; the OPS5 rule-based representation also allows the incorporation of new specification features. These two issues are discussed in the final part of this chapter.

Chapter IV is devoted to protocol validation using the OPS5 production system. A brief introduction to protocol validation approaches, i.e., global state reachabil-
Figure 23: Overview of the protocol development process using the OPS5 production system
ity analysis and deductive inference, is given. The formal modeling of global state reachability analysis using the OPS5 production system is explained and exemplified. Referring to Figure 23, production rules are used to formally describe the generation of all reachable global states. Logical errors and logical properties in communication protocols, such as unspecified receptions, deadlock, channel overflow, nonexecutable interaction, quiescent state, and ambiguous state [78], are also formally represented in production rules. Additional elements are used to describe global states, logical errors and logical properties and other frequently changed information in protocol validation [52]. The motivation and the explanation of incremental validation is presented in detail with the corresponding representation in production rules. Finally, we discuss the pros and cons of global state reachability analysis, and advantages and disadvantages of protocol validation using the OPS5 production system.

Chapter V is devoted to protocol implementation using the OPS5 production system. The main consideration, i.e., the machine-independent part and the machine-dependent part, for protocol implementation is given. The general implementation model, including how to formally model the machine-independent part, how to formulate the machine-dependent part, and the control flow, based on the OPS5 production system is explained and exemplified. Referring to Figure 23, the machine-independent part is formally modeled by the protocol designer in the specification phase. The machine-dependent part, including external procedures and interface data structures used for interlayer communication, is realized according to the operational architecture and the host operating system in the implementation phase. The comparison
of the OPS5 production system with other FDTs is presented. Finally, we also dis-
cuss the possibility of using the rule-based computation formalism to formulate the
machine-dependent part and the code process paradigm of the OPS5 production sys-
tem.

Chapter VI is devoted to protocol testing using the OPS5 production system.
The rationale of the incremental test method, which is avoidance of repeated execu-
tion of some transitions in the communication protocol under testing, is discussed in
detail. The traditional test approaches, including test sequence generation methods
and test architectures, are briefly introduced. Then, our incremental test approach is
presented. The incremental test sequence generation algorithm is presented in detail
with the corresponding formal modeling in production rules. Referring to Figure 23,
production rules are used to generate the test sequences and to control the status
of IUT. Some additional elements are used to describe pseudo communication rules,
state monitoring and status backtracking, and the remaining elements are used in the
external procedures that communicate with UT, LT and the Test Management Cen-
ter. The prototypes of the modified test architectures are explained. Two issues in
the incremental test approach, which are the resolution of the synchronization prob-
lem and the reduction of coordinating effort between clients and the test laboratory,
are explained. Finally, we discuss the main disadvantage, i.e., additional software in
the local side, and the corresponding compensation of the incremental test approach.
7.2 Conclusions and Future Research Issues

In this dissertation, a computer-aided protocol design methodology based on the OPS5 production system has been proposed. Based on a globally shared dataspace (working memory) in which different types and levels of information are all represented in a uniform structure, and based on the integration of both rule-based and procedure-based computation formalisms into a single system, the OPS5 production system can be used as a single representation mechanism for all of the four phases in the protocol design process. Through the use of the OPS5 representation mechanism, the translators used between the heterogeneous representation mechanisms can be eliminated; through the globally shared dataspace, any two phases can be interconnected by combining the related elements, and the incremental process can be applied to the validation phase and the test phase.

The current Formal Description Techniques (FDT), such as ESTELLE [130], LOTOS [131] and SDL [132], are devoted to formally describing the functional specifications of communication protocols. In the first decade of FDTs development, it has been proved that FDTs are needed and useful. However, since most design effort is focused on the issues of how to formally model functional specifications of communication protocols, the validation, implementation, and test phases of protocol design become nontrivial work for protocol engineers [81]. Moreover, even though new FDTs are proposed and used in protocol specification, the principles used for protocol validation and protocol testing are still based on the old principles. These drawbacks make FDTs not easily accepted for practical use by protocol engineers. In order to
make FDTs more acceptable and to simplify the effort needed in designing communication protocols, the next generation of FDTs is expected to provide more powerful facilities so that they can be used in the four phases of protocol design. One example of such facilities is the functionality used for formally describing the knowledge of protocol validation, such as global state reachability analysis; another example is the functionality used for formally describing the knowledge of protocol testing, such as the test sequence generation.

In the future, there are still a lot of research issues that can be done based on the proposed computer-aided protocol design methodology. The possible future research issues are described as follows:

1. Extensions to PDPS:

   - **PDPS with an intelligent interface:** In order to obtain better performance, an intelligent interface should be added to PDPS to classify protocols and to apply a suitable global state reduction strategy to each kind of protocol. In this way, through an intelligent protocol validation system that is executed in a multiprocessor environment, global state reachability analysis of more complicated protocols can be carried out.

   - **PDPS for performance analysis:** In order to obtain optimal values of parameters in communication protocols, communication protocol performance analysis [67] can be realized by adding probability and time constraints to each communication rule.
• PDPS for N-entity protocols: Current PDPS is based on 2-entity protocols. To consider N-entity protocols, PDPS should be extended by generalizing all of the elements and production rules.

2. Evaluation of the OPS5-based protocol implementation: A comparison of the OPS5-based implementation with other approaches, such as manual implementation and other (semi-)automatic implementation approaches, can be done.

3. The incremental test method: A test laboratory for testing communication protocols can be constructed according to the proposed incremental test method.

4. From the OPS5 production system to other FDTs: Formal reasoning about protocols requires some sort of formal description of the protocol and its properties. Different formal methods should provide different mechanisms for reasoning. As mentioned previously, current FDTs are devoted to formally describing functional specifications of communication protocols. In order to modify current FDTs so that they are more powerful, researchers in Protocol Engineering can try to apply the concepts in the OPS5 production system to extend FDTs' functionalities.

5. Protocol conversion: Protocol conversion provides interoperability between two different network protocols [43, 63, 74]. Most existing methods for protocol conversion require human ingenuity to derive a conversion specification (such as conversion seed) [10, 16, 17, 84, 85], either from the protocol specification or the service specification. In order to have an automatic derivation of conversion
specifications, the semantics of protocol specifications or service specifications should be formally represented. For example, the semantics can be represented in OPS5 production rules or other Artificial Intelligence (AI) techniques. In this way, existing protocol conversion approaches [19, 20, 99, 100, 101, 118, 119, 120] may be able to mechanically derive conversion specifications. However, some heuristics should be provided to narrow down the search for conversion specifications from the given protocol specifications or service specifications.
Appendix A

FORMAL SPECIFICATION OF THE FULL DUPLEX ALTERNATING BIT PROTOCOL

In this appendix, a full duplex transport Alternating Bit Protocol (ABP) [69] in which communications are carried over an unreliable medium is formally modeled in OPS5 production rules. When an entity receives a data transmission request from the user, a sequence number 0 or 1 is appended to the data block; then, this data block is transmitted one or more times until a positive acknowledgment is received. A timer is used to detect whether any loss occurs during the transmission and a time-out period should be set suitably to properly inform the transmission loss. On the other hand, when an entity receives a data reception request from the user, a data block will be delivered in a FIFO order to the user. An entity also always stays ready to receive an incoming message, either a data block or an acknowledgment, from the network layer. If the incoming data block has the expected sequence number, this data block is stored in the corresponding receive buffer and an acknowledgment is sent to the send entity. Otherwise, the receive entity responds with an acknowledgment for this data block to the send entity only. For simplicity, we assume the acknowledgment message will not be corrupted or garbled.
A.1  Formal Specification in OPS5 Production Rules

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; external procedures and vector attributes declaration ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(external trandata)
(external delidata)
(external tranack)
(external trannegack)
(external initiate)
(vector-attribute data)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; elements declaration ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(literalize Inform

Aentity ; IDs of ABPs
N.Jpoint ; abp_network interaction points
U.Jpoint ; abp_user interaction points
state ; current state of a ABP
Tseq ; current send sequence number
Rseq          ; expected receive sequence number
Time)         ; estimated transmission time

(literalize commrule
  Ipoint       ; interaction points
  mess         ; message type
  cstate       ; current state
  nstate)      ; next state

(literalize T_DataRequest
  U_Ipoint     ; abp_user interaction points
  type         ; message type
  data)        ; data block

(literalize DataIndication
  N_Ipoint     ; abp_network interaction point
  seq          ; receive sequence number
  type         ; receive message type
  data)        ; data block

(literalize DataRequest
  Aentity      ; IDs of ABPs
  N_Ipoint     ; abp_network interaction point
  timer        ; time_out period
  seq          ; transmission sequence number
  data)        ; data block
(literalize T_DataIndication
  U_Ipoint ; abp_user interaction point
data) ; data block
(literalize R_data
  N_Ipoint ; abp_network interaction point
data) ; data block
(literalize T_data
  N_Ipoint ; abp_network interaction point
data) ; data block
(literalize timeout
  Aentity ; IDs of ABPs
  N_Ipoint) ; abp_network interaction point
(literalize no_user
  total) ; no. of total users

;;;;; production rules ;;;

(p Transmit_Data_Request
  {(T_DataRequest "U_Ipoint <U> "type transmission "data <m> ) <buf>}
(Inform "Aentity <A> "N_Ipoint <N> "U_Ipoint <U> "state <s1> "Tseq <sn>
  "Time <t>)}
(commrule "Ipoint user "mess T_D_R "cstate <s1> "nstate <s2>)

- ->

(make Tdata "N_Ipoint <N> "data (substr <buf> data inf))

(call trandata DataRequest "Aentity <A> "N_Ipoint <N> "timer <t> "seq <n> 
"data (substr <buf> data inf))

(modify 2 "state <s2>)

(remove 1))

(p Receipt_Data_Request

(T_DataRequest "U_Ipoint <U> "type reception)

{ (Rdata "N_Ipoint <N> "data <m>) <buf> }

(Inform "Aentity <A> "N_Ipoint <N> "U_Ipoint <U> "state <s1>)

(commrule "Ipoint user "mess R_D_R "cstate <s1> "nstate <s2>)

- ->

(call delidata T_DataIndication "U_Ipoint <U> "data (substr <buf> data inf))

(modify 3 "state <s2>)

(remove 1 2))

(p Receive_Data

{ (DataIndication "N_Ipoint <N> "seq <sn> "type trandata "data <m>) <buf> }

(Inform "Aentity <A> "N_Ipoint <N> "U_Ipoint <U> "state <s1> "Rseq <sn>)

(commrule "Ipoint server "mess R_D "cstate <s1> "nstate <s2>)}
(call tranack DataRequest "Aentity <A> "N_Jpoint <N> "seq <sn> "data (substr <buf> data inf))
(modify 2 "Rseq (compute 1 - <sn>))
(modify 2 "state <s2>)
(remove 1))

(p Receive_error_data
(DataIndication "N_Jpoint <N> "seq <n1> "type trandata "data <m>)
(Inform "Aentity <A> "N_Jpoint <N> "U_Jpoint <U> "state <s1> "Rseq {<n2> <> <n1>}})
(commrule "Ipoint server "mess R_E_D "cstate <s1> "nstate <s2>)
- - >
(call trannegack DataRequest "Aentity <A> "N_Jpoint <N> "seq <n1>)
(modify 2 "state <s2>)
(remove 1))

(p Receive_Acknowledgement
(DataIndication "N_Jpoint <N> "seq <sn> "type tranack)
(Inform "Aentity <A> "N_Jpoint <N> "U_Jpoint <U> "state <s1> "Tseq <sn>)
(Tdata "N_Jpoint <N> "data <m>)
(commrule "Ipoint server "mess R_A "cstate <s1> "nstate <s2>)
(modify 2 ~Tseq (compute 1 - <sn>))
(modify 2 ~state <s2>)
(remove 1 3))

(p Receive_Time_out
(timeout ^Aentity <A> ^N_Jpoint <N>)
(Inform ^Aentity <A> ^N_Jpoint <N> ^U_Jpoint <U> ^state <s1> ^Tseq <sn> ^Time <t>)
{(Tdata ^N_Jpoint <N> ^data <m>) <buf>}
(commrule ^Ipoint server ^mess time_out ^cstate <s1> ^nstate <s2>)

- ->
(call transdata DataRequest ^Aentity <A> ^N_Jpoint <N> ^timer <t> ^seq <sn> ^data (substr <buf> data inf))
(modify 2 ~state <s2>)
(remove 1))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;; initialize the initial WM configuration ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(p set_up
(no_user "total \{t > 0\})
->
(make Inform "Aentity <t> "N_\text{point} <t> "U_\text{point} <t> "state init "Tseq 0 "Rseq 0 "time n)
(modify 1 "total (compute <t> - 1)))

(p set_down
(no_user "total \{t = 0\})
->
(remove 1)
(make task waiting)
(call initiate initial)) ; initialize the interface processes and start to work

(p wait_loop
(task waiting)
->
(remove 1)
(make task waiting)
(call initiate newcycle)) ; go to a waiting status to wait for the other events

(p insertion
(start) ; start is a reserved element to indicate the starting execution
A.2 Explanation

The above specification consists of three parts: the first part declares external procedures and vector attributes; the second part declares the formats of elements; and the third part describes the production rules.

Five external procedures, \texttt{trandata}, \texttt{delidata}, \texttt{tranack}, \texttt{trannegack} and \texttt{initiate}, are used to extract the values in the interface elements (abstract service primitives) and generate the corresponding outgoing and local events according to the interface data structures.

Production rule \texttt{Transmit\_Data\_Request} is used for receiving a transport data re-
quest service primitive of data transmission. Element $T_{DataRequest}$, which represents the abstract transport data request service primitive, is inserted by the system process that administers the receptions from the upper layer. Element $Inform$ records some variables: attribute $Tseq$ records the current transmission sequence number of this local entity and attribute $Time$ indicates the time-out period. Another attribute, $Rseq$ (which indicates the currently expected receive sequence number) in element $Inform$, is not expressed explicitly, because it is not referenced in this production rule. Element $estate$ records the current state of a local entity and the identifiers of the local entity ($Aentity$), the local entity’s network interaction point ($N_Ipoint$) and the local entity’s user interaction point ($U_Ipoint$). Element $Tdata$ is a send buffer that stores the currently transmitted data block to be acknowledged, where function (substr $<buf>$ data inf) extracts values from attribute $data$ to the end field (denoted as $inf$) of element $T_{DataRequest}$ (which is designated by "<buf>"). Element $DataRequest$ represents the abstract network data request service primitive. This rule is applied when local entity $<A>$ is in state $<s1>$ and receives a message of data transmission from user interaction point $<U>$. External procedure $trandata$ will invoke the corresponding system process to issue this transmission event to the network layer and to initialize a timer to monitor this transmission event. After that, the state of this local entity is updated to $<s2>$.

Production rule $Receipt_{Data Request}$ is used for receiving a transport data request service primitive of data reception. Element $Rdata$ records the data block to be delivered and is inserted by the system process that administers the receptions from
the network layer. Element $T.DataIndication$ represents the abstract transport data indication service primitive. This rule is applied when local entity $<A>$ is in state $<s1>$ and receives a message of data reception from user interaction point $<U>$. External procedure $delidata$ will invoke the corresponding system process to issue the deliver event to the served user.

Production rule $Receive.Data$ is used for receiving an abstract network data indication service primitive of normal data. Element $DataIndication$, which represents the abstract network data indication service primitive, is inserted by the system process that administers the receptions from the network layer. To express the predicate that the receive sequence number is equal to the currently expected one, the same identifier is used in attribute $seq$ of element $DataIndication$ and in attribute $Rseq$ of element $Inform$. This rule is applied when local entity $<A>$ is in state $<s1>$ and receives a data block with the expected sequence number from network interaction point $<N>$. External procedure $tranack$ will invoke the corresponding system process to transmit an acknowledgment to the peer entity and the corresponding system process that administers the receive data blocks to indicate the readiness for upper users' reception. Then, the next expected receive sequence number in attribute $seq$ of element $Inform$ is calculated by using function $compute$. Five arithmetic operators (addition, subtraction, multiplication, division and modulus) used in communication protocols are all supported.

Production rule $Receive.Error.Data$ is used for receiving an abstract network data indication service primitive of abnormal data. This production rule is similar to the
previous one used in the normal data reception, except for the expression of receiving an out of sequence data block. To express the predicate that the receive sequence number is different from the expected one, the receive sequence number $n1$ should be different from the expected one $n2$. (i.e., the expression $\{<n1> <> <n2>\}$ in attribute $Rseq$ of element $Inform$.) External procedure $trannegack$ will invoke the corresponding system process to transmit an acknowledgment to the peer entity.

Production rule $Receive\_Acknowledgement$ is used for receiving an abstract network data indication service primitive of acknowledgment. This rule is applied when a local entity receives an acknowledgment message from the peer entity.

Production rule $Receive\_Time\_out$ is used for receiving a time-out primitive. Element $timeout$, which represents the abstract primitive of a time-out event (in network interaction point $<N>$), is inserted by the system process that administers the timer for message transmission. This production rule is applied when a local entity receives a time-out abstract primitive.

In the initialization, production rules $set\_up$ and $insertion$ initialize the initial configuration of the WM. Production rule $set\_up$ also sets up the connection of interaction modules, an ABP entity with a user interaction point and an ABP entity with a network interaction point. After the WM is initialized, production rule $set\_down$ calls an external procedure to initialize the execution environment. This procedure forks three more concurrent processes: one administers the incoming events from the upper layer; the second one administers the incoming events from the lower layer; and the third one administers the local environment events (like timer). The original process
administers the OPS5 rule-based specification and outgoing events invocation. After initialization, all of these processes are in waiting status.
Appendix B

EXAMPLES OF EXECUTING PDPS

In this appendix, illustrations of executing PDPS are presented. For convenience, the validation process of the protocol depicted in Figure 24 is used as an example.

![Diagram of protocol execution]

Figure 24: The protocol used for illustration in this chapter
B.1 The Options Provided by PDPS in the Beginning

What is the channel bound ? 1
What is the initial state for entity 1 ? a
What is the initial state for entity 2 ? a
Do you want to specify the final states ? Y or N ? n
Print all error messages immediately ? Y or N ? y
Print the whole states step by step ? Y or N ? n
Stop when an error is found or after all errors are found ? 1 or all ? all
Print the error path immediately after an error is found ? Y or N ? n
Which definition is a reception error in your version ?
1: limit to a receiving state only
2: do not limit to a receiving state
1 or 2 ? 1
B.2 The Options Provided by PDPS after Validation: Listing all Rules

Add or delete rules \(A\) or \(D\)?
List all rules or states \(R\) or \(S\)?
List a rule or a state \(SR\) or \(SS\)?
List a path to a state ? \(P\)
List all errors ? \(E\)?
List all quiescent & ambiguous states ? \(Q\)?
List all final states ? \(F\)?
List all redundant rules ? \(RR\)?
List all duplicated states ? \(DS\)?
List all states created by a special rule? \(RS\)?
The END ? END ? r

rule id = 13 entity = 2 type = r cstate = c message = 1 nstate = d
rule id = 12 entity = 2 type = r cstate = c message = 2 nstate = a
rule id = 11 entity = 2 type = r cstate = b message = 1 nstate = c
rule id = 10 entity = 2 type = r cstate = a message = 1 nstate = c
rule id = 9 entity = 2 type = s cstate = d message = 4 nstate = c
rule id = 8 entity = 2 type = s cstate = a message = 3 nstate = b
rule id = 7 entity = 1 type = r cstate = c message = 3 nstate = a
rule id = 6 entity = 1 type = r cstate = d message = 3 nstate = c
<table>
<thead>
<tr>
<th>Rule ID</th>
<th>Entity</th>
<th>Type</th>
<th>CState</th>
<th>Message</th>
<th>NState</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>r</td>
<td>b</td>
<td>4</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>r</td>
<td>b</td>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>r</td>
<td>a</td>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>s</td>
<td>b</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>s</td>
<td>a</td>
<td>1</td>
<td>b</td>
</tr>
</tbody>
</table>

Finish!
B.3 The Options Provided by PDPS after Validation: Listing all States

Add or delete rules? A or D?
List all rules or states? R or S?
List a rule or a state? SR or SS?
List a path to a state? P
List all errors? E?
List all quiescent & ambiguous states? Q?
List all final states? F?
List all redundant rules? RR?
List all duplicated states? DS?
List all states created by a special rule? RS?
The END? END? s

ID = 34
< c > 2
   empty < b >

ID = 33
< b > empty
   empty < b >
ID = 31
< c > 1
empty < b >

ID = 30
< c > empty
empty < c >

ID = 28
< b > empty
3 < c >

ID = 26

< a > 1 , 2
empty < b >

ID = 22
< b > empty
empty < c >
ID = 20
< c > 1, 2
empty < b >

ID = 19
< b > 1
empty < b >

ID = 18
< a > empty
empty < b >

ID = 17
< c > empty
3 < b >

ID = 16
< c > empty
empty < a >

ID = 15
< c > 2
empty < c >

ID = 14
< c > 1, 2
empty < a >

ID = 12
< b > 1
empty < a >

ID = 11
< b > 2, 1
empty < c >

ID = 10
< a > 2
empty < c >

ID = 7
< c > empty
3, 3 < b >
ID = 6
  < c > empty
  3 < a >

ID = 5
  < c > 2
  3 < c >

ID = 4
  < c > 1, 2
  3 < b >

ID = 3
  < b > 1
  3 < b >

ID = 2
  < a > empty
  3 < b >

ID = 1
  < a > empty
empty < a >

Total state number: 35

duplicated state number: 11

different state number: 24
B.4 The Options Provided by PDPS after Validation: Listing all Errors

Add or delete rules ?A or D ?
List all rules or states ? R or S ?
List a rule or a state ? SR or SS ?
List a path to a state ? P
List all errors ? E?
List all quiescent & ambiguous states ? Q ?
List all final states ? F ?
List all redundant rules ? RR?
List all duplicated states ? DS ?
List all states created by a special rule? RS?
The END ? END ? e

state id = 34 : reception error in entity 2 !
state id = 30 : a deadlock state !
state id = 26 : channel overflow error in queue 1->2 !
state id = 20 : channel overflow error in queue 1->2 !
state id = 14 : channel overflow error in queue 1->2 !
state id = 11 : channel overflow error in queue 1->2 !
state id = 7 : channel overflow error in queue 2->1 !
state id = 4 : channel overflow error in queue 1->2 !
Finish!
B.5 The Options Provided by PDPS after Validation: Listing a State

Add or delete rules ?A or D ?
List all rules or states ? R or S ?
List a rule or a state ? SR or SS ?
List a path to a state ? P
List all errors ? E?
List all quiescent & ambiguous states ? Q ?
List all final states ? F ?
List all redundant rules ? RR?
List all duplicated states ? DS ?
List all states created by a special rule? RS?
The END ? END ? ss

What is the state ID ? 30

ID = 30
< c > empty
empty < c >
B.6 The Options Provided by PDPS after Validation: Listing the Path to a State Using State IDs

Add or delete rules? A or D?
List all rules or states? R or S?
List a rule or a state? SR or SS?
List a path to a state? P
List all errors? E?
List all quiescent & ambiguous states? Q?
List all final states? F?
List all redundant rules? RR?
List all duplicated states? DS?
List all states created by a special rule? RS?
The END? END? p

What is the state ID? 30
Print out the state IDs or the states from state ID = 30? I or S? 1

⇒ 1 ⇒ 2 ⇒ 3 ⇒ 28 ⇒ 30
B.7 The Options Provided by PDPS after Validation: Listing the Path to a State Using States

Add or delete rules? A or D?
List all rules or states? R or S?
List a rule or a state? SR or SS?
List a path to a state? P
List all errors? E?
List all quiescent & ambiguous states? Q?
List all final states? F?
List all redundant rules? RR?
List all duplicated states? DS?
List all states created by a special rule? RS?
The END? END? p

What is the state ID? 30
Print out the state IDs or the states from state ID = 30? I or S? s
=>

ID = 1
< a > empty
empty < a >
=>
ID = 2
< a > empty
3 < b >

ID = 3
< b > 1
3 < b >

ID = 28
< b > empty
3 < c >

ID = 30
< c > empty
   empty < c >
B.8 An Example of the Incremental Validation: Adding a Rule

Add or delete rules? A or D?
List all rules or states? R or S?
List a rule or a state? SR or SS?
List a path to a state? P
List all errors? E?
List all quiescent & ambiguous states? Q?
List all final states? F?
List all redundant rules? RR?
List all duplicated states? DS?
List all states created by a special rule? RS?
The END? END? a

Add a rule for entity 1 or 2? Finish adding? List all rules so far?
1 or 2 or F or R? 1

For sending or receiving a message? s
From which state? c
What is the message? 2
To which state? c
Are you sure? Y or N? y

A new rule!

A deadlock error associated with state id = 30 is deleted!

Add a rule for entity 1 or 2? Finish adding? List all rules so far?
1 or 2 or F or R? f

Finish adding rules!
B.9 An Example of the Execution of PDPS after Adding a New Rule

Type "Y" after Add or Delete rules to see the changes in states

Find more errors ? Y or N ?

Add or delete ? A or D ?

List a specified rule or a state ? SR or SS ?

List all rules or all states so far ? R or S ?

Want to change options ? C ? y

Continue!

RECEPTION ERROR IN ENTITY 2 ! State ID = 36

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 36

state ID = 37 is the SAME as state ID = 20

state ID = 38 is the SAME as state ID = 15

RECEPTION ERROR IN ENTITY 2 ! State ID = 39

RECEPTION ERROR IN ENTITY 2 ! State ID = 40
CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 40

RECEPTION ERROR IN ENTITY 2 ! State ID = 41

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 41

RECEPTION ERROR IN ENTITY 2 ! State ID = 42

RECEPTION ERROR IN ENTITY 2 ! State ID = 43

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 43

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 45

state ID = 46 is the SAME as state ID = 39

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 47

state ID = 48 is the SAME as state ID = 44

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 50
CHANNEL OVERFLOW IN Queue 1→2 ! State ID = 51

RECEPTION ERROR IN ENTITY 2 ! State ID = 52

CHANNEL OVERFLOW IN Queue 2→1 ! State ID = 52

RECEPTION ERROR IN ENTITY 2 ! State ID = 53

RECEPTION ERROR IN ENTITY 2 ! State ID = 54

CHANNEL OVERFLOW IN Queue 1→2 ! State ID = 54

RECEPTION ERROR IN ENTITY 2 ! State ID = 55

CHANNEL OVERFLOW IN Queue 1→2 ! State ID = 55

RECEPTION ERROR IN ENTITY 2 ! State ID = 56

state ID = 57 is the SAME as state ID = 36

state ID = 59 is the SAME as state ID = 53
CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 60

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 61

state ID = 62 is the SAME as state ID = 49

CHANNEL OVERFLOW IN Queue 1->2 ! State ID = 63

state ID = 64 is the SAME as state ID = 58

Finish create all states!
B.10 An Example of the Incremental Validation: Deleting a Rule

Add or delete rules? A or D?

List all rules or states? R or S?

List a rule or a state? SR or SS?

List a path to a state? P

List all errors? E?

List all quiescent & ambiguous states? Q?

List all final states? F?

List all redundant rules? RR?

List all duplicated states? DS?

List all states created by a special rule? RS?

The END? END? d

DELETE a rule for entity 1 or 2? Finish deleting? List all rules?

1, 2, F or R? 1

For sending or receiving a message s

From which state? b

What is the message? 2

To which state? c
Are you sure? Y or N? y

Delete a rule ID = 2

* delete a state ID = 29 and it is a duplicated state!
* delete a state ID = 23 and it is a duplicated state!
* delete a state ID = 34

delete a reception error in entity 2 associated with state id = 34

* delete a state ID = 20

* delete a state ID = 21 and it is a duplicated state!

delete a channel overflow error in Q1->2 associated with state id = 20

* delete a state ID = 14

* delete a state ID = 15

* delete a state ID = 16

delete an ambiguous state in entity 2 existed between state ID = 16 & 30

* delete a state ID = 17

* delete a state ID = 18

delete an ambiguous state in entity 1 existed between state ID = 18 & 33

* delete a state ID = 19

* delete a state ID = 22 and it has other duplicated state id = 24

Copy back such a state id = 24!

delete an ambiguous state in entity 2 existed between state ID = 22 & 33

delete an ambiguous state in entity 1 existed between state ID = 22 & 30

delete a quiescent state associated with state id = 22
delete an ambiguous state in entity 2 existed between state ID = 18 & 1
delete a quiescent state associated with state id = 18
delete an ambiguous state in entity 1 existed between state ID = 16 & 1
delete a quiescent state associated with state id = 16
delete a channel overflow error in Q1->2 associated with state id = 14
* delete a state ID = 4
* delete a state ID = 26
* delete a state ID = 27 and it is a duplicated state!
delete a channel overflow error in Q1->2 associated with state id = 26
* delete a state ID = 5
* delete a state ID = 10
* delete a state ID = 25 and it is a duplicated state!
* delete a state ID = 11
* delete a state ID = 12 and it has other duplicated state id = 35
Copy back such a state id = 35!
* delete a state ID = 13 and it is a duplicated state!
* delete a state ID = 24
delete a channel overflow error in Q1->2 associated with state id = 11
* delete a state ID = 6
* delete a state ID = 9 and it is a duplicated state!
* delete a state ID = 7
* delete a state ID = 8 and it is a duplicated state!
delete a channel overflow error in Q2->1 associated with state id = 7
delete a channel overflow error in Q1->2 associated with state id = 4
Finish deleting states !

DELETE a rule for entity 1 or 2 ? Finish deleting ? List all rules ?
1, 2, F or R ? 2

For sending or receiving a message r
From which state ? c
What is the message ? 2
To which state ? a
Are you sure ? Y or N ? y
Delete a redundant rule ID = 12

DELETE a rule for entity 1 or 2 ? Finish deleting ? List all rules ?
1, 2, F or R ? f

Finish deleting rules !

Deadlock error ! State ID = 33
B.11 An Example of the Execution of PDPS after Deleting a Rule

Type "Y" after Add or Delete rules to see the changes in states

Find more errors? Y or N?

Add or delete? A or D?

List a specified rule or a state? SR or SS?

List all rules or all states so far? R or S?

Want to change options? C? y

Continue!

state ID = 36 is the SAME as state ID = 3

It is an ambiguous state ID = 37 in entity 2 with state ID = 33

It is an ambiguous state ID = 37 in entity 1 with state ID = 30

Deadlock error! State ID = 37

It is a quiescent state! State ID = 37

Finish create all states!
Appendix C

COMPARISON OF THE TRADITIONAL TEST SEQUENCE GENERATION METHOD WITH THE INCREMENTAL TEST SEQUENCE GENERATION METHOD

This appendix presents a comparison of the traditional test sequence generation method (using the UIO method) and the incremental test sequence generation method. The formal modeling of the incremental test sequence generation is also described in Section C.2. For convenience, the transmitter section of the half duplex Alternating Bit Protocol (ABP) [92] is used in this appendix.

C.1 Comparison between the UIO Method and the Incremental Method

Figure 25 shows the transmitter section of the half duplex ABP specified in CFSM, where "?" represents an input and "!" represents an output [92].

Table 14 shows the UIO sequences generated for this half duplex ABP. Figure 26 shows the corresponding optimized test sequences generated for weak conformance testing by using the UIO method (Table 14 and Figure 26 are in [92]). In Figure 26, "ri" represents a reset input that moves IUT from any state to the initial state.
Figure 25: The transmitter section of the half duplex ABP specified in CFSM

Table 14: The unique input/output sequences for the protocol in Figure 25

<table>
<thead>
<tr>
<th>state</th>
<th>UIO sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>user?data/null, null/net!data0</td>
</tr>
<tr>
<td>1</td>
<td>null/net!data0</td>
</tr>
<tr>
<td>2</td>
<td>null/timer!start, timer?timeout/null, null/net!data0</td>
</tr>
<tr>
<td>3</td>
<td>timer?timeout/null, null/net!data0</td>
</tr>
<tr>
<td>4</td>
<td>user?data/null, null/net!data1</td>
</tr>
<tr>
<td>5</td>
<td>null/net!data1</td>
</tr>
<tr>
<td>6</td>
<td>null/timer!start, timer?timeout/null, null/net!data1</td>
</tr>
<tr>
<td>7</td>
<td>timer?timeout/null, null/net!data1</td>
</tr>
</tbody>
</table>
Figure 26: The optimized test sequences generated for weak conformance testing by using the UIO method
WEAK CONFORMANCE TESTING:

STRONG CONFORMANCE TESTING:

Figure 27: The test sequence generated by using the incremental method

Figure 27 shows the incremental test sequences, including sequences for both weak conformance testing and strong conformance testing, of the half duplex ABP. For simplicity, the same notation as that in Figure 26 is used.

C.2 Formal Modeling of the Incremental Test Sequence Generation for the Half Duplex ABP

The formal specification of the incremental test sequence generation for the half duplex ABP is as follows.

1. Set the current state $S_c$ and the entrance state $S_e$ of IUT as the initial state $S_i$.

   This can be represented by the following production rule.

   (p Sequence.Generation_Initial
   (initial_status "state <is> "seq <q>))
(Inform "state <cs>)
(store_inform "state <is>)
- >
(modify 2 "state <is> )
(modify 3 "seq <q> )
(make enter_state "state <is>))

where element initial\_status records the initial state of IUT, element enter\_state records the entrance state of IUT after the application of the transition to be tested, and element store\_inform records IUT's status information, i.e., values of the variables recording the dynamic information of the protocol entity.

2. Nondeterministically choose an unexplored transition \( S_c \xrightarrow{I/O} S_n \). Input I is applied and the output is checked to verify whether it is 0. This can be represented by the following production rule:

\[
(p \text{ Sequence\_Generation}
(\text{Inform } "\text{state } <\text{cs}>\n(\text{enter\_state } "\text{state } <\text{es}>\n\{(\text{state\_stack } "\text{s\_stk } <\text{st}> \} <\text{stk}>\})
(\text{transition } "\text{in\_point } <\text{ip}> "\text{in\_mess\_type } <\text{imt}> "\text{in\_mess } <\text{im}> "\text{out\_point} <\text{op}> "\text{out\_mess\_type } <\text{omt}> "\text{out\_mess } <\text{om}> "\text{current\_state } <\text{cs}> "\text{next\_state } <\text{ns}>\n- >
\]
(modify 2 ^state <ns>)
(modify 3 ^s_stk <cs> (substr <ststk> 2 inf))
(call event_generation event ^in_point <ip> ^in_mess_type <imt>
 ^in_mess <im> ^out_point <op> ^out_mess_type <omt> ^out_mess <om>)
(removed 4)

where element state_stack records the order of traversed states. In order to have
the depth-first process, the first-in-last-out stack-like access strategy is used to
store the traversed states in the second action. Element event contains I/O
messages and I/O interaction points of the transition to be tested. External
procedure event_generation with element event will activate UT or LT to cre­
ate the corresponding input message to trigger IUT and to record the output
message from IUT.

3. Verify whether the entrance state of IUT is correct or not. When IUT has the
expected entrance state, the status of IUT is recorded. The status information,
e.g., values of the variables recording the status of the protocol entity, should
be recorded because IUT's status needs to be rollbacked when the test sequence
is backtracked to this entrance state again. When IUT has an unexpected
entrance state, IUT is erroneous. The state verification and status recording
are realized by the following production rules:
(p Entrance_State_Matched
  (Inform \("state <s> \"seq <q>\)
  (enter_state \("state <s>\)
  (store_inform \("state <s>\)
  - ->
  (modify 3 \("seq <q>\)
  (call state_report state_verification \("result matched\))

(p Entrance_State_Unmatched
  (Inform \("state <cs>\)
  (enter_state \{"es\ <> <cs>\})
  - ->
  (call state_report state_verification \("result unmatched\)

where external procedure state_report with element state_verification will activate the test management center to log the result. Element store_inform stores the status information of IUT when IUT is state <s>.

4. If there is an unexplored transition originating from state \(S_n\), then keep \(S_n\) as the current state and apply the production rule in step 2; otherwise, search backward to find an ancestor state \(S_a\) of \(S_n\) from which some unexplored transitions originate. When state \(S_a\), from which some unexplored transitions originate, is found, IUT's status needs to be rollbacked to the corresponding \(S_a\)'s status. The following production rule describes the state backtracking and status rollback:
In production rule State_Backtrack, in order to express the condition that the current state is out of unexplored transitions, the same variable is used in attributes state of element Inform and current_state of element transition. In the RHS, the first action changes the current state of IUT to the ancestor state. The second action pops out the head item from element state_stack. This production rule is repeatedly applied until a state from which some unexplored transitions originate is found. Then the production rule in step 2 is applied to continue the test process.
In production rule \textit{Status Rollback}, in order to express the condition that the current state has some unexplored transitions and to find the corresponding status information, the same variable is used in attributes \textit{state} of elements \textit{Inform} and \textit{store.inform}, and attribute \textit{current.state} of element \textit{transition}. In the RHS, element \textit{Inform}, which records IUT’s status information, e.g., variables, is reset to the expected one.

5. When state $S_a$, from which some unexplored transitions originated, does not exist, this implies that all transitions have been explored and this test session can be terminated. The following production rule describes the termination condition.

\begin{verbatim}
(p Protocol.Test.Over
 (state_stack \^s.stk \{<st> = nil\})
 - ->
 (call event_generation test_over))
\end{verbatim}

When element \textit{state_stack} contains no traversed state, this means that all of the transitions have been tested. External procedure \textit{event_generation} with element \textit{test_over} notifies the test management center to terminate the test session.
BIBLIOGRAPHY

Production Systems on Message Passing Computers: Simulation Results and Analysis.

An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours.


A Grammar-Based Methodology for Protocol Specification and Implementation.

Automated Protocol Implementation with RTAG.


Mechanical Verification and Automatic Implementation of Communication Protocols.

Semiautomatic Implementation of Communication Protocols.

Design Principles for Communication Gateways.

Protocol Specification for OSI.

On Communication Finite-State Machines.

On Integrating Logic Programming and Database.


Deriving A Protocol Converter.

An Improved Protocol Test Sequence Generation Based on UIOs.


An Approach to Protocol Complementation for Interworking.

Minimal Length Test Sequences for Protocol Conformance.

Approaches Utilizing Segment Overlap to Minimize Test Sequences.

Synchronizable Protocol Test Generation via the Duplex Technique.

Testing Software Design Modeled by Finite-State Machines.

Global State Graph Reduction Techniques for Protocol Validation in the EFSM Model.

Test Case Generation for Protocols Specified in Estelle.
[27] A. Chung and D. P. Sidhu.  
On Conditions for Defining a Closed Cover to Verify Progress for Communicating  
Finite State Machines.  
1989.  

Fault Coverage of Probabilistic Test Sequence.  

Rapid Prototyping of Communication Protocols Using A New Parallel Language.  

How Could Estelle Become a Better FDT?.  

Toward Formal Verification of Rule-based Systems: A Shared Dataspace Perspective.  
Louis, Missouri, 1989.  

A UNITY-Style Programming Logic for Shared Dataspace Programs.  

[33] J.D. Day.  
The OSI Reference Model.  

[34] M. Diaz.  
Modeling and Analysis of Communication and Cooperation Protocols Using Petri  
Net Based Models.  

Artificial Intelligence.  
RETÉ: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.

Protocol Verification Using Database Technology.

[38] G. Gonenc.

The Proof of a Distributed System in Gypsy.

Protocol Validation by Maximal Progress State Exploration.

Synthesis of Communicating Finite-State Machines with Guaranteed Process.

Protocol Validation by Fair Progress State Exploration.

Protocol Conversion.

Derivation of Test Cases for LAP-B from a LOTOS specification.

Suitability of Message Passing Computers for Implementing Production Systems.


[55] C. M. Huang, Y. I. Chang, and M. T. Liu.

[56] T. Ishida.
Optimizing Rules in Production System Programs.

[57] T. Ishida.
Methods and Effectiveness of Parallel Rule Firing.

An Organizational Approach to Adaptive Production Systems.
Proceedings of the 8th National Conference on Artificial Intelligence, pp. 52-58, 1990.

Introduction to Expert Systems.


Protocol Verification Via Projections.

[63] S. S. Lam.
Protocol Conversion.
A Relational Algebraic Approach to Protocol Verification.


[66] F. J. Lin, P. M. Chu, and M. T. Liu.


Verification of CSP Programs Based on Communication Assertions.

The Features and Facilities of ESTELLE.

[70] R. J. Linn.


On the Complexity of Generating the Test Sequence Without Synchronization Problem.

[73] M. T. Liu.
Protocol Engineering.
[74] M. T. Liu.  
Protocol Conversion.  


A Methodology for Specifying and Analyzing Communication Protocols and Services.  


[78] C. S. Lu.  
Automated Validation of Communication Protocols.  


Protocol verification: the First Ten Years, the Next Ten Years, invited paper.  

Fault Detection for Sequential Machines by Transition Tours.  

[84] K. Okumura.
A Formal Protocol Conversion Method.

[85] K. Okumura.
Generation of Proper Adapters and Converters from a Formal Specification.

[86] B. Pehrson.
Protocol Verification for OSI.

[87] T. F. Piatkowski.

The Derivation of Conformance Tests from LOTOS Specifications.

[89] D. Rayner.
OSI Conformance Testing.

[90] G. C. Roman.
Language and Visualization Support for Large-Scale Concurrency.

Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency.

A Protocol Test Generation Procedure.

A Test Design Methodology for Protocol Testing.
[94] B. Sarikaya.
Conformance Testing: Architectures and Test Sequences.

Executable Communication Rule Systems.

From State machines to Temporal Logic: Specification Methods for Protocol Standards.


A Software Design Method and Its Application to Protocol and Communication Software Development.

[99] J. C. Shu and M. T. Liu.
A Synchronization Model for Protocol Conversion.

Protocol Conversion between Complex Protocols.

An Approach to Indirect Protocol Conversion.

Executable Logic Specifications for Protocol Service Interfaces.


Implementing OSI Systems.


Specification Technique.

A LOTOS-Based Development Strategy.

Optimal Test Sequence Generation for Protocols: the Chinese Postman Algorithm Applies to Q.931.

Formal Specification in OSI.


Generalized Technique for Communication Protocol Validation.

Protocol Validation by Random State Exploration.

Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping.

A Modular Approach to Constructing Protocol Converters.

A Parallel Model for Constructing Protocol Converters.

[120] Y. W. Yao and M. T. Liu.
Constructing Protocol Converters with Guaranteed Service.

An Improved Method for Automatic Validation of Communication Protocols.
[122] M. C. Yuang.
Survey of Protocol Verification Techniques Based on Finite State Machine Models.


Parallel Protocol Verification Using the Two-phase Algorithm.

[125] H. X. Zeng and D. Rayner.
The Impact of the Ferry Concept on Protocol Testing.

Promoting the 'Local' Test Method with the New Concept 'Ferry Clip'.

New Advances in Ferry Testing Approaches.

On Ferry Clip Approaches in Protocol Testing.


ESTELLE - A Formal Description Technique Based on Extended State Transition Model.