INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript 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. These are also available as one exposure on a standard 35mm slide or as a 17” x 23” black and white photographic print for an additional charge.

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.
Towards automating protocol synthesis and analysis

Chu, Peil-Ying Mark, Ph.D.
The Ohio State University, 1989

Copyright ©1989 by Chu, Peil-Ying Mark. All rights reserved.
Towards Automating Protocol Synthesis and Analysis

DISSERTATION

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

By

Peil-Ying Mark Chu, B.S.E.E., M.S.

*****
The Ohio State University

1989

Dissertation Committee
Professor Ming T. Liu
Professor T. H. Lai
Professor N. Soundararajan

Approved by

[Signature]
Adviser
Department of Computer and Information Science
Copyright by
Peil-Ying Mark Chu
1989
To My Parents and Family
ACKNOWLEDGMENTS

I am in deep appreciation of my adviser Professor Ming T. Liu for introducing me into the research area of Protocol Engineering and for guiding me while this work is being done. I am also grateful to Professors Teng-Hwang Lai and Neelam Soundararajan for agreeing to serve as members of my reading committee and for their efforts in reviewing the manuscript.

My parents deserve the greatest merits and special thanks. They have been always very supportive during my graduate study at the Ohio State University. Abundant credits should go to my loving wife Shun-Tsu Hsu, whose patience and understanding have made this work possible.

I am greatly indebted to my office mate and dearest friend Fuchun Joseph Lin for his encouragement. All the members of ACS (Advanced Communication Systems) research group have my sincere appreciation for providing valuable suggestions and comments on this work.

The work reported in this dissertation was supported by U.S. Army CECOM, Fort Monmouth, New Jersey, under Contract Nos. DAAB07-87-K-A005 and DAAB07-88-K-A003. The views, opinions and/or findings contained in this dissertation are those of the author and should not be construed as an official Department of Defense of the Army position, policy, or decision. We should like to express our thanks to Dr. Charles J. Graff for funding the project and for his encouragement and support.
VITA

Jun. 2, 59
Born: Taichung, Taiwan, Republic of China

Sep. 77 - Jun. 81
B.S.E.E., Department of Electrical Engineering,
National Taiwan University, Taipei, Taiwan

Sep. 83 - Mar. 85
M.S., Department of Computer and Information Science,
The Ohio State University, Columbus, Ohio

Mar. 85 - Mar. 89
Ph.D., Department of Computer and Information Science,
The Ohio State University, Columbus, Ohio

PUBLICATIONS

Protocol Synthesis

   (co-author: M. T. Liu)

Protocol Analysis


Protocol Specification

FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Architecture and Distributed Systems:
Dr. Ming T. Liu

Studies in Distributed Algorithms:
Dr. T. H. Lai

Studies in Programming Languages:
Dr. N. Soundararajan
# Table of Contents

Dedication ................................................................. ii  
Acknowledgments ....................................................... iii  
Vita ........................................................................ iv  
List of Figures ........................................................... xi  

CHAPTER

I  
Introduction ............................................................... 1  
1.1 Protocol Synthesis and Analysis ................................. 2  
1.2 Objectives and Main Results ..................................... 3  
1.3 Chapter Summary .................................................. 5

II  
Service Concept ......................................................... 8

III  
A Survey of Protocol Synthesis Techniques ...................... 14  
3.1 Synthesis Techniques Requiring No Service Specification 15  
3.1.1 Zafiropulo’s Reception Production Rules ................. 17  
3.1.2 Sidhu’s Protocol Design Rules ................................. 17  
3.1.3 Zhang’s Protocol Synthesis Algorithm ...................... 18
APPENDICES

A  Syntax of the ETG Protocol Specification Language  128

B  Diagnosis Report on Running the ETG Protocol Validation System for the Alternating Bit Protocol  139

C  Application of the CTL Model Checking Algorithm to Live Variable Analysis  158
List of Figures

1.1 Various aspects of protocol engineering. ...................................................... 7

2.1 Architectural model for layered protocol design. ................................. 9

2.2 A network architecture based on the OSI Reference Model. ............... 10

2.3 From the (N)-service specification and the (N-1)-service specification to the (N)-protocol specification. .......................................................... 13

3.1 Protocol derivation algorithm and error-recovery transformation. ......... 25

4.1 Architecture for two-entity protocols and services. .............................. 27

4.2 Specification of a two-slot bounded buffer communication service. ....... 29

4.3 Specification of the simplified ISO transport service. ......................... 30

4.4 Reachability graph of the service FSMs in Fig. 4.2. ............................. 36

4.5 Reachability graph of the service FSMs in Fig. 4.3. ............................. 37

4.6 Illustration of Step 1 part for nonconcurrent transitions. ..................... 44

4.7 Illustration of Step 1 part for concurrent transitions. ........................... 48

4.8 Adjustment of the initial state pointer. .................................................. 51

4.9 Transformation of a concurrent transition for deriving the protocol FSM for Entity 1. ................................................................. 53
8.2 A possible use of our protocol derivation algorithm and global state
graph reduction techniques. ........................................ 122

A.1 Syntax of the ETG protocol specification language. .................. 136

C.1 Transforming a transition in an FSM in the EFSM model to two con­
secutive transitions in a finite-state reference-assignment system. ... 161
CHAPTER I

Introduction

Being the backbone of distributed computing systems, communication protocols play an indispensable role in keeping smooth operation of the communication networks that underlie the distributed systems. In OSI (Open Systems Interconnection) Reference Model [56], a communication protocol is a set of rules governing the interaction and coordination between a number of communicating entities at a certain protocol layer. The purpose of these rules is to provide some intended communication services to the service users, which constitute the next higher layer above the concerned one.

Correctness of communication protocols is critical to reliable operation of computer networks. However, due to the concurrent and nondeterministic nature of communicating entities, the behavior of most communication protocols exhibits a high degree of complexity, making the assurance of correctness of communication protocols difficult. Consequently, the correctness problem of communication protocols has long
been recognized as an important issue and many research efforts have been directed to ensure the correctness of communication protocols.

1.1 Protocol Synthesis and Analysis

Protocol synthesis and analysis are two inherently different but complementary approaches to ensuring the correctness of communication protocols [52]. In the synthesis approach, rules ensuring some desirable properties are enforced during the protocol design process. In the analysis approach, an already designed protocol is first examined to reveal some properties, desirable or undesirable, and then modified to get rid of the undesirable ones.

The synthesis approach has the advantage over the analysis approach in aiding protocol designers to reduce the possibility of making errors, if not to prevent it totally, during the protocol design process. However, protocol analysis (validation and verification) is still inevitably playing an important role in developing correct protocols even though there are already some protocol synthesis techniques proposed in the literature. This is because every protocol synthesis technique has its own limitations imposed by the underlying model, and has its own scope restrained by the enforced communication patterns and the error-preventing power of the technique itself. For
instance, some protocol synthesis techniques need protocol analysis to go hand in hand for detecting protocol logical errors that are not preventable by the protocol synthesis techniques. The above reasons warrant the co-existence of research on both protocol synthesis and on protocol analysis, which are the subject of this research.

1.2 Objectives and Main Results

The first part of this research is with regard to automating the process of deriving protocol specifications from service specifications. However, the protocol derivation process for an arbitrary communication service appears to be formidably difficult. As a result, we concentrate on a class of communication services whose behavior can be described by a set of directly coupled Finite State Machines (FSMs). For a service specified in this state-transition model, we provide a protocol derivation algorithm which produces the protocol specification from the given service specification automatically once some additional information about the decision options and initiation option is provided by the protocol designer. The provision of the additional information is to ensure that the produced protocol specification is indeed intended by the protocol designer.

In enhancing our algorithm to deal with error-prone communication media, we
further devise an error-recovery transformation procedure. The error-recovery transformation procedure transforms any protocol specification produced from the protocol derivation algorithm into the specification of an error-recoverable protocol that can handle the error-prone media.

Since every protocol synthesis technique, including ours, has its own limitations imposed by the underlying model and assumptions, protocol analysis (validation and verification) is still needed to ensure the correctness of protocols beyond the expressive power of the underlying model. One of the most effective and mechanizable protocol validation/verification techniques is reachability analysis. However, the global state explosion problem restricts the applicability of reachability analysis to protocols whose global state graphs are not huge. Consequently, global state graph reduction techniques are needed for alleviating this problem. In the second part of this research, we propose two such techniques for protocol specifications in the Extended Finite State Machine (EFSM) model. The impact of the first global state graph reduction technique on the incremental protocol validation is also examined.

1.3 Chapter Summary

The organization of the remainder of this dissertation is as follows. Chapter II to
Chapter V deal with the first part of this research and Chapter VI to Chapter VII with the second part.

Chapter II introduces the service concept in the context of communication protocols. The importance of the service concept in protocol design is addressed herein.

Chapter III provides a brief survey of protocol synthesis techniques in the literature. Two categories are identified, namely synthesis techniques that require service specification to be initially given and synthesis techniques that do not require service specification to be initially given. Some arguments about why protocol synthesis techniques in the second category are the right way to go are then stated. The drawbacks of each protocol synthesis technique in the second category are scrutinized to motivate the formation of our protocol derivation algorithm.

Chapter IV elaborates our protocol derivation algorithm. The model for service specifications and protocol specifications is given, followed by the detailed presentation of the three steps of the algorithm.

Error-recovery transformation is the topic of Chapter V, which demonstrates the three transformation rules in the error-recovery transformation procedure. The sink-state and duplicate acceptance problems would occur in a transformed protocol specification; therefore, a fix must be performed on the transformed protocol specification.
to get rid of these problems. The problems and the suggested remedies are then discussed.

Chapter VI turns our attention to the protocol analysis issues, using the Extended Finite State Machine (EFSM) as an underlying model. Reachability analysis is then used to validate/verify protocol specifications in the EFSM model.

In Chapter VII, we propose two global state graph reduction techniques. The first reduction technique is examined more carefully than the second one. One upper bound of the effect of the technique on the reduction of the global state graph is derived and the impact of the technique on the incremental protocol validation is discussed.

Finally, Chapter VIII summarizes the main results and the contribution made by this research to the state of the art in Protocol Engineering [34]. Fig. 1.1 shows the block diagram for various aspects of protocol engineering, and the shaded regions are of our concern in this research, namely protocol synthesis to derive protocol specifications from given service specifications and protocol analysis to examine protocol specifications for revealing some desirable or undesirable properties. After the summary of the main results of this research is given, the limitations of our approaches to protocol synthesis and analysis are itemized and suggestions for future work are made.
Figure 1.1. Various aspects of protocol engineering.
CHAPTER II

Service Concept

Service concept is receiving more and more attention in current communication protocol design [47]. With the abstraction facility in service concept, the complexity problem of protocol design can be alleviated to such an extent that protocol designers are capable of dealing with it competently. Employing the service concept, the architectural model for layered protocol design in the OSI (Open Systems Interconnection) world [56] is elegant and succinct (see Fig. 2.1). The OSI Reference Model is developed by the International Standards Organization (ISO). Based on several principles, seven layers are formed in the OSI Reference Model. Fig. 2.2 shows a network architecture that is based on the OSI Reference Model.

The architecture model shown in Fig. 2.1 may be considered as an abstraction of the network architecture shown in Fig. 2.2. In a specific protocol layer, say the Nth layer, the communicating entities together provide a set of capabilities to the service
users through the (N)-Service Access Points (or (N)-SAPs for short) by obeying the (N)-protocol and by making use of the service provided by the layers below this one. In other words, the (N)-protocol combined with the service provided by lower layers forms a service provider to the service users, which may be end users or the communicating entities in the next higher layer, i.e., the (N+1)th layer. Consequently, the (N)-protocol can be regarded as the logical implementation of the (N)-service given the (N-1)-service available for use. Since the (N)-SAPs are the only places through which the (N)-service can be accessed to the service users, the internal mechanism embedded in the protocol and the interaction between communicating entities are not visible to the service users. For example, the Alternating Bit Protocol (ABP) [2] provides a service that guarantees the correct transfer of data in sequence from one
Figure 2.2. A network architecture based on the OSI Reference Model.
user to the other. However, the use of an alternating bit variable in each communicating entity and the retransmission mechanism in the communicating entity serving the user that has data for transmission are not visible to both service users.

The set of capabilities provided by the communicating entities in a protocol layer is presented by the execution of a group of well-defined service primitives. A service primitive is considered as an elementary interaction between a service user and the service provider during which certain values for the various parameters of the primitive are established to which both user and provider can refer. Thus each (N)-service primitive is associated with an (N)-SAP and executed at that (N)-SAP. The specification of an (N)-service can be expressed in terms of the possible orderings of service primitives associated with the (N)-SAPs and their parameter values dependencies [47]. On the other hand, the specification of the (N)-protocol can be expressed in terms of the possible orderings of service primitives associated with the (N)-SAPs and the (N-1)-SAPs and their parameter values dependencies.

In the following, several advantages of the service concept are discussed. These are based on arguments given in [47].

1. The main advantage of utilizing service concept in communication protocol design is to provide a framework on which the complexity of protocol design can be better managed.
2. A protocol designed using service concept can be changed without affecting any layer other than the one the protocol resides in. This is due to the principle of separation of concerns in service concept.

3. Yet another advantage of using service concept in protocol design is to facilitate the correctness proofs. Without the service concept, the verification of a communication system becomes an unsurmountably difficult task.

In the next chapter, a list of protocol synthesis techniques will be surveyed. Most of them concentrate on the interaction between communicating entities. Such protocol synthesis techniques do not treat the service concept formally and resort to protocol designers for the responsibility of taking the service and semantics of protocols into account. We believe, however, the right approach to protocol design should be one that treats the service concept formally. In particular, we feel that one should start from a formal specifications of the (N)-service and the (N-1)-service to construct the desired formal specification of the (N)-protocol, as depicted by Fig. 2.3. The next chapter will give more detailed explanation of the motivation of this approach.
Figure 2.3. From the (N)-service specification and the (N-1)-service specification to the (N)-protocol specification.
CHAPTER III

A Survey of Protocol Synthesis Techniques

Protocol synthesis techniques are few compared to protocol analysis techniques published in the literature. The area of protocol synthesis appears to be less developed than the area of protocol analysis. In this chapter, we will briefly survey protocol synthesis techniques, the list of which is by no means exhaustive; but we simply give an overview of some protocol synthesis techniques, the goals they have achieved, and their limitations. With the limitations of the surveyed techniques in mind, we will provide the motivation for developing our own protocol synthesis technique at the end of the chapter.

To classify protocol synthesis techniques, two attributes are identified, viz., the initial provision of service specification and the degree of human intervention. For the first attribute, some protocol synthesis techniques require the provision of service speci-
ification initially but others don't. For the second attribute, some protocol synthesis techniques require human intervention (called interactive or incremental techniques) and others don't (called fully automated techniques). In the following, we first classify protocol synthesis techniques according to the first attribute. The surveyed techniques in the first category can be further divided into interactive ones and fully automated ones, whereas those in the second category are all fully automated.

3.1 Synthesis Techniques Requiring No Service Specification

Protocol synthesis techniques in this category do not require the initial existence of a service specification to which the synthesized protocol specification has to conform. Therefore, the protocol designer himself/herself is responsible for the semantics of the synthesized protocol specification. The goal of these techniques is to construct protocol specifications free from the following logical errors: nonspecified reception, nonexecutable interaction, deadlock, unboundedness, and improper termination. Each technique has achieved either a portion or the whole of the goal. Generally speaking, the techniques achieving just a portion of the goal have higher flexibility than those achieving the whole of the goal.
The aforementioned logical errors are likely to occur in any protocol and not related to the specific functions provided by the protocol. Therefore, they are sometimes called syntactic errors of the protocol. The logical errors of a protocol are defined informally as follows.

- **nonspecified reception**: a message reception can occur but not specified.

- **nonexecutable interaction**: a specified interaction is not executable for any possible execution path of the protocol.

- **deadlock**: Every communicating entity is waiting for messages while all the channels connecting them are empty.

- **unboundedness**: the protocol may reach a situation in which the numbers of messages in some channels are unbounded.

- **improper termination**: the protocol may reach a global state from which no predefined final global state is reachable.

Seven techniques are surveyed in this section, the first four and the last one of which are interactive, and the fifth and sixth of which are fully automated. Each of them is briefly described in the following.
3.1.1 Zafiropulo's Reception Production Rules

Zafiropulo et al. proposed three reception production rules, which are used in an interactive protocol synthesis system [52]. As long as these rules are obeyed, two protocol logical errors, unspecified reception and nonexecutable interaction, can be prevented for any synthesized protocol specification. These rules, however, are only applicable to two-entity protocols. To handle multi-entity protocols, they later proposed a different set of production rules [8], which are much more complicated than those for two-entity protocols. Protocol logical errors such as deadlock, though not preventable, may be monitored by the system in the process of designing a protocol. The internal representation of protocol behavior in the system is $N$ trees for an $N$-entity protocol.

3.1.2 Sidhu's Protocol Design Rules

Sidhu proposed four protocol design rules, which can be used to monitor all kinds of protocol logical errors [42]. However, the protocol designer has to specify all the interactions (message transmissions and receptions) between communicating entities. Thus the technique is just an algorithm to validate a protocol in the process of designing it and not a real synthesis technique. The internal representation of protocol behavior
in the technique is a global state transition graph.

3.1.3 Zhang’s Protocol Synthesis Algorithm

Zhang et al. proposed a protocol synthesis algorithm consisting of three production rules and two deadlock avoidance rules [54, 53]. Like Sidhu’s protocol design rules, the internal representation of protocol behavior in Zhang’s protocol synthesis algorithm is a global state transition graph. Zhang’s technique can be considered as an improvement over Sidhu’s technique in that it enhanced Sidhu’s technique by automatically generating the specifications of all receptions that can occur and by adding deadlock avoidance rules to prevent possible occurrence of deadlock. Zhang’s technique is restricted to two-entity protocols and it is suspected that the deadlock avoidance rules are not general enough to cover all deadlock-free two-entity protocols.

3.1.4 Choi’s Sequence Method

Choi presented a method for constructing protocol specifications in the Finite State Machine (FSM) model by first synthesizing a pair of regular expressions of star height zero or one and then converting the regular expressions to equivalent FSMs [9]. His method can prevent all kinds of protocol logical errors mentioned in the beginning of
this section. However, his technique is limited to two-entity protocols whose entity FSMs correspond to regular expressions of star height at most one.

### 3.1.5 Gouda’s Synthesis Algorithm

Given a partial specification of a communicating entity, the algorithm proposed by Gouda et al. enforces a fixed communication pattern between two communicating entities in order to construct the complete protocol specification in which all kinds of design errors are not existent [16]. One disadvantage of this algorithm is that the generated specification for the peer entity is just one of the possible correct specifications and may not be the one intended by the protocol designer. Furthermore, the algorithm is applicable only to two-entity protocols.

### 3.1.6 Ramamoorthy’s Automated Protocol synthesizer

Ramamoorthy et al. proposed an automated protocol synthesizer that makes use of six transformation rules to build up the specification for the peer entity from a given specification for the local entity [37]. All kinds of design errors can be prevented by this synthesizer if the specification for the local entity possesses some desirable properties. The synthesizer suffers the same drawbacks as Gouda’s algorithm.
3.1.7 Kakuda’s Component-Based Synthesis

Kakuda et al. generalized Ramamoorthy’s six rules to come up with twenty
two patterns of components, which may be used to construct multi-entity protocols[21]. Moreover, Kakuda’s technique allows interaction with the protocol designer to
increase flexibility for protocol construction. All kinds of protocol logical errors can
be prevented.

3.2 Synthesis Techniques Requiring Service Specification

Protocol synthesis techniques in this category requires the initial provision of a
service specification to which the synthesized protocol specification has to conform.
The goal of these techniques is not only to construct protocols that are free from the
protocol logical errors, but also to mandate that the synthesized protocol specification
conform to the given service specification. In the sequent, we briefly describe three
such techniques and their disadvantages.
3.2.1 Merlin's Submodule Construction Method

Merlin et al. proposed a method of determining the specification for the missing entity from a given service specification and the specifications for the remaining entities [30]. Unfortunately, the technique does not guarantee the deadlock-freedom for the synthesized protocol specification and thus must be supplemented by an analysis procedure to detect the deadlock.

3.2.2 Prinoth's Protocol Construction Algorithm

The input to Prinoth's protocol construction algorithm [35] is actually a specification refined from a service specification by adding some auxiliary action transitions and the output from the algorithm is a protocol specification. Therefore, the protocol designer himself/herself has to refine the service specification to produce the input to the algorithm. The algorithm itself does not include a method to perform the refinement of the service specification.
3.2.3 Bochmann's Protocol Derivation Algorithm

Bochmann et al. proposed an algorithm to derive a protocol specification from a given service specification [7]. A service in their model is described by an expression of service primitives connected by sequence, parallelism, and alternative operators. A syntax tree is employed to collect the necessary information for the send and receive actions required for synchronizing service primitives. Consequently, their specification language is not able to describe a service containing an infinite number of possible execution paths. Inclusion of a recursion operator, as suggested in their paper, may remove the deficiency but may also complicate their algorithm to some extent.

3.3 Motivation of Our Protocol Derivation Algorithm

The seven protocol synthesis techniques in the first category provide some rules or methods for obtaining the complete protocol specification, starting from a partial protocol specification, either interactively or fully automatically. However, they don't have a service specification initially given as a reference. The protocol designer himself/herself is responsible for the semantics of the synthesized protocol specification; therefore, he/she must resort to his/her intuitive understanding of the intended service, a very informal task in current protocol design. As a result, more work is needed at
the stage of protocol verification.

The three protocol synthesis techniques in the second category do consider service specifications in a formal manner. Merlin's work, however, additionally requires the existence of specifications for \( (n-1) \) communicating entities, where \( n \) is the number of communicating entities in the protocol layer of interest. Prinoth's work and Bochmann's work are more ambitious since only the service specification of the interested layer is needed at the outset. Nevertheless, in Prinoth's work, some auxiliary actions (similar to the synchronization messages discussed in Chapter IV) are, in some cases, needed to be added into the service specification prior to the application of his protocol construction algorithm; but the algorithm does not provide a method to help designers perform the refinement of the service specification by including such auxiliary actions. In Bochmann's work, the required synchronization messages are derived automatically; however, the service specification language is not able to express a service containing an infinite number of possible execution paths. In our protocol derivation algorithm, we essentially follow the same approach taken by Bochmann at al., thus inheriting the advantages of their approach. But we use a state-transition model, which can easily describe a service containing an infinite number of possible execution paths by using transition loops in FSMs and which seems to be a more natural and better understood model.
In Chapter IV and Chapter V, we will elaborate on our protocol derivation algorithm and error-recovery transformation procedure, respectively. As illustrated in Fig. 3.1, the protocol derivation algorithm will produce a protocol specification from a given service specification, assuming the communication media are error-free; and the error-recovery transformation procedure will add error-recovery capability to the protocol specification produced from the first stage, with the assumption that the communication media are error-prone. The error-recovery transformation procedure contains three transformation rules as well as rules for fixing sink-state problem and duplicate acceptance problem. Both the protocol derivation algorithm and the error-recovery transformation procedure can be fully automated.
3 transformation rules and rules for fixing sink-state problem and duplicate acceptance problem

Figure 3.1. Protocol derivation algorithm and error-recovery transformation.
CHAPTER IV

Our Protocol Derivation Algorithm

In this Chapter we present an algorithm for deriving the protocol specification from a given service specification, which is described by a set of directly coupled Finite State Machines (FSMs) [3]. These FSMs together regulate the execution sequence of service primitives as intended by the service specification. Throughout the chapter, only two-entity protocols are considered and the algorithm presented is geared toward this class of protocols. However, we believe the generalization of the algorithm to multi-entity protocols is straightforward as long as the definition of decision options (to be discussed in Section 4.2) for multi-entity protocols is made appropriately.

This chapter proceeds as follows. Section 4.1 explains the model we use to specify services and protocols. Section 4.2 presents our protocol derivation algorithm for producing the protocol specification from a given service specification, assuming the underlying communication medium is free from transmission errors.
4.1 The Model

We are concerned about the architecture for two-entity protocols and services as shown in Fig. 4.1, which is an instantiation of that shown in Fig. 2.1. User 1 (User 2) is served by Entity 1 (Entity 2) at SAP 1 (SAP 2). The underlying communication medium service between these two entities is assumed to be two unidirectional queues, which are FIFO and error-free. Later in Chapter V we will relax the 'error-free' assumption when we come to the subject of error-recoverable protocols.

4.1.1 Service Specifications

To facilitate easier construction, better readability and understandability of service specifications, the division of a service specification into two parts, namely a local
constraint and a global constraint, has been suggested in the literature [4, 11]. A local constraint restricts the relative execution order of service primitives as observed at each of the SAPs individually, while a global constraint further restricts the relative execution order of service primitives associated with different SAPs. Following this suggestion, a service specification in our model is composed of $\alpha$ (where $\alpha \geq 0$) local constraint FSMs $M_1, \ldots, M_\alpha$ and $\beta$ (where $\beta \geq 0$) global constraint FSMs $N_1, \ldots, N_\beta$, where $\alpha$ and $\beta$ cannot be both zero. A local constraint FSM may only refer to service primitives associated with a specific SAP. On the other hand, a global constraint FSM must refer to service primitives associated with different SAPs. Both kinds of FSMs are referred to as service FSMs. These service FSMs together, through direct coupling [3], cooperatively restrict the behavior of the service provider as intended by the service specification. By direct coupling of FSMs, we mean that all the FSMs containing a transition labeled $P$ must participate in the action for the action $P$ to occur. Two examples will be used throughout this chapter and the next. The first example is a two-slot bounded buffer communication service, whose specification in our model is shown in Fig 4.2. This example contains two local constraint FSMs $M_1$ and $M_2$, and two global constraint FSMs $N_1$ and $N_2$. The second example is the connection establishment and release phases of the simplified ISO transport service, as specified using Communicating Sequential Processes (CSP) in [27], without the provider-initiated disconnections. In this service specification (see Fig. 4.3) there are five service FSMs: two local constraint FSMs, $M_1$ and $M_2$; and three global constraint
We define a function $Sap(P)$ that returns the identity of the SAP with which the service primitive $P$ is associated. A particular state in a service FSM is indicated as the initial state and a set of states designated as the final state set. We assume that every state in these service FSMs is reachable via the directly-coupled interaction among them; otherwise, we can always remove the unreachable state without affecting the semantics of the specification. A communication service can be either nonterminating or terminating, depending on whether the final state sets of the service FSMs are all empty or all nonempty. If it is found that the final state sets of some service FSMs are empty and some are not, the service specification must be wrong. Therefore, it is clear that the two-slot bounded buffer communication service (see Fig. 4.2) is a
Figure 4.3. Specification of the simplified ISO transport service.

<table>
<thead>
<tr>
<th>Service primitive</th>
<th>P</th>
<th>Sap(P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creq</td>
<td>Connection request (from User 1)</td>
<td>SAP_1</td>
</tr>
<tr>
<td>Cind</td>
<td>Connection indication (to User 2)</td>
<td>SAP_2</td>
</tr>
<tr>
<td>Cres</td>
<td>Connection response (from User 2)</td>
<td>SAP_2</td>
</tr>
<tr>
<td>Cconf</td>
<td>Connection confirmation (to User 1)</td>
<td>SAP_1</td>
</tr>
<tr>
<td>Dreq1</td>
<td>Disconnect request from User 1</td>
<td>SAP_1</td>
</tr>
<tr>
<td>Dind2</td>
<td>Disconnect indication to User 2</td>
<td>SAP_2</td>
</tr>
<tr>
<td>Dreq2</td>
<td>Disconnect request from User 2</td>
<td>SAP_2</td>
</tr>
<tr>
<td>Dind1</td>
<td>Disconnect indication to User 1</td>
<td>SAP_1</td>
</tr>
</tbody>
</table>

⇒ : initial state pointer
○ : final state
nonterminating one, whereas the simplified ISO transport service (see Fig. 4.3) is a terminating one.

It is natural to allow the coordinated concurrent execution of service primitives at different SAPs in addition to the strictly sequential order expressible by the conventional FSM. Therefore, *concurrent transitions* are added into our service model, appearing in some global constraint FSMs. A concurrent transition is a transition with a label in a form of \([R_{11}, \ldots, R_{1t} \parallel R_{21}, \ldots, R_{2u}]\), where \(Sap(R_{11}) = SAP_1, i = 1, \ldots, t, \) and \(Sap(R_{2j}) = SAP_2, j = 1, \ldots, u\); the label means that one of the service primitives on the left hand side of ‘\(\parallel\)’ may execute concurrently with one of those on the right hand side. A concurrent transition can be started once its head service state is reached. The concurrent transition ends when the two selected service primitives from \(R_{1i}, i = 1, \ldots, t, \) and from \(R_{2j}, j = 1, \ldots, u, \) respectively, are executed to completion and its tail service state is reached. The tail node of a concurrent transition may only carry either \(<1>\) or \(<2>\) as its decision option since the decision option \(<1|2>\) assigned to its tail node makes no sense for the concurrent transition due to the uncertainty of which entity executes a service primitive most recently if the concurrent transition is executed. To make the distinction clearer, service primitive transitions mentioned in the previous paragraph will also be called *nonconcurrent transitions*. 
In passing, it is worthwhile to mention a model called STOCS (Synchronous TOnken based Communicating State) model proposed by Garg [12]. After restricting the concurrent transitions and final states in our service specification model, we have got a model which is a special case of STOCS model that allows only a single token in each communicating state machine. Consequently, the STOCS model is more powerful than the restricted model. However, it appears to be rather difficult to do protocol synthesis based on the STOCS model.

4.1.2 Consistency of Service Specifications

Using a set of directly coupled FSMs to describe a communication service, protocol designers may inadvertently write inconsistent service specifications. A procedure for checking the consistency of a service specification is thus required. A service specification is consistent if and only if

it is properly terminated in the case of terminating services, or

it is deadlock-free in the case of nonterminating services.

A straightforward way of checking the consistency of a service specification is to construct the reachability graph of the directly coupled service FSMs and decide
whether proper termination or deadlock freedom holds in the reachability graph. The
reachability graph actually defines the control structure of the service specification,
which is finite since the service FSMs are all finite and directly coupled with one
another. Thus the consistency check of a service specification is decidable. The reachabil-
ity graph $G$ of a set of directly coupled service FSMs $M_1, \ldots, M_\alpha, N_1, \ldots, N_\beta$, is
defined as follows:

1. $[s_0^1, \ldots, s_0^\alpha, r_1^0, \ldots, r_\beta^0]$ is an initial node in $G$, where $s_0^1, \ldots, s_0^\alpha, r_1^0, \ldots, r_\beta^0$
   are the initial states of $M_1, \ldots, M_\alpha, N_1, \ldots, N_\beta$, respectively.

2. If
   
   (a) $S = [s_1, \ldots, s_\alpha, r_1, \ldots, r_\beta]$ is a node in $G$; and
   
   (b) there are nonconcurrent transitions all with an identical label, say
   $P$, from states $s_{i_1}, \ldots, s_{i_p}, r_{j_1}, \ldots, r_{j_q}$, to $s'_{i_1}, \ldots, s'_{i_p}, r'_{j_1}, \ldots, r'_{j_q}$,
   respectively, where $1 \leq i_1 < i_2 < \ldots < i_p \leq \alpha$, and $1 \leq j_1 < j_2 < \ldots < j_q \leq \beta$; and
   
   (c) $P$ is not the label of any transition in $M_k$ for $k \in \{1, \ldots, \alpha\} -
   \{i_1, \ldots, i_p\}$, and in $N_l$ for $l \in \{1, \ldots, \beta\} - \{j_1, \ldots, j_q\}$;
   
   then
   
   $S' = [s_1', \ldots, s'_{i_1}, \ldots, s'_{i_p}, \ldots, s_\alpha', r_1', \ldots, r'_{j_1}, \ldots, r'_{j_q}, \ldots, r_\beta']$
   
   is a node in $G$ and there is a transition from $S$ to $S'$ labeled
3. If

(a) \( S = [s_1, \ldots, s_\alpha, r_1, \ldots, r_\beta] \) is a node in \( G \); and

(b) there is a concurrent transition labeled \([R_{11}, \ldots, R_{1t} \parallel R_{21}, \ldots, R_{2u}]\),

from state \( r_j \) to \( r'_j \), \( 1 \leq j \leq \beta \), transitions all with an identical label, say \( R_{1k}, 1 \leq k \leq t \), from state \( s_{i_1}, \ldots, s_{i_p} \) to \( s'_{i_1}, \ldots, s'_{i_p} \), respectively, transitions all with an identical label, say \( R_{2l}, 1 \leq l \leq u \), from state \( s_{m_1}, \ldots, s_{m_q} \) to \( s'_{m_1}, \ldots, s'_{m_q} \), respectively, where

\[ 1 \leq i_1 < i_2 < \ldots < i_p \leq \alpha, \ 1 \leq m_1 < m_2 < \ldots < m_q \leq \alpha, \]

and \( \{i_1, \ldots, i_p\} \cap \{m_1, \ldots, m_q\} = \emptyset \); and

(c) \( R_{1k} \) and \( R_{2l} \) are not the label of any transitions in \( M_h \) for \( h \in \{1, \ldots, \alpha\} - \{i_1, \ldots, i_p\} - \{m_1, \ldots, m_q\} \);

then

\[ S' = [\ldots, r_1, \ldots, r'_j, \ldots, r_\beta] \]

is a node in \( G \) and there is a transition from \( S \) to \( S' \) labeled \([R_{1k} \parallel R_{2l}]\).

(Note that the first ellipsis in \( S' \) stands for the states of local constraint FSMs, among which \( s_{i_1}, \ldots, s_{i_p}, s_{m_1}, \ldots, s_{m_q} \) are changed to \( s'_{i_1}, \ldots, s'_{i_p}, s'_{m_1}, \ldots, s'_{m_q} \); otherwise unchanged.)
The above three items exactly define the nodes and transitions in the reachability graph of the service FSMs. There is a requirement to be met by the reachability graph $G$ in order for the proposed protocol derivation algorithm (to be presented in Section 4.2) to derive a deadlock-free protocol specification. The third item of the above definition needs to require that "if there is a concurrent transition labeled $[R_{11}, \ldots, R_{1t} \parallel R_{21}, \ldots, R_{2u}]$ from state $r_j$ to $r'_j$, it must be executable".

After the reachability graph $G$ is constructed according to the above definition, the consistency of the service specification can be checked for proper termination and deadlock freedom. Proper termination and deadlock freedom are defined based on the reachability graph $G$ as below.

Proper termination holds in $G$ if and only if the following predicate is true:

If $S$ is a non-final node in $G$, then there exists a sequence of transitions from $S$ to a final node $F = \{f_1, \ldots, f_\alpha, g_1, \ldots, g_\beta\}$, where $f_1, \ldots, f_\alpha$, $g_1, \ldots, g_\beta$ are one of the final states of $M_1, \ldots, M_\alpha, N_1, \ldots, N_\beta$, respectively.

Deadlock freedom holds in $G$ if and only if the following predicate is true:

If $S$ is a non-final node in $G$, then $S$ has at least one outgoing transition.
Note that proper termination in $G$, according to the above predicates, implies deadlock freedom in $G$; therefore, deadlock freedom is not an additional requirement of consistency in the case of terminating services. In the case of nonterminating services, $G$ being deadlock-free amounts to the statement that every node in $G$ has at least one outgoing transition. The algorithms presented in [28] can be modified to find the reachability graph and to check proper termination and deadlock freedom.

For example, to check the consistency of the service specifications given in Fig. 4.2 and Fig. 4.3, respectively, their reachability graphs as shown in Fig. 4.4 and Fig. 4.5, respectively, are obtained. From Fig. 4.4, it is seen that the service specification for the two-slot bounded buffer communication service is consistent since, every node in the graph has at least one outgoing transition. From Fig. 4.5, we can see that the
* Nodes with an underlined label are final nodes.

Figure 4.5. Reachability graph of the service FSMs in Fig. 4.3.

Service specification for the simplified ISO transport service is consistent too, since from every non-final node in the graph there exists at least a sequence of transitions to reach a final node.

4.1.3 Protocol Specifications

A protocol specification consists of two entity specifications, each of which may contain a single FSM or several directly coupled FSMs (called protocol FSMs). Any protocol FSM for Entity $E$, $E = 1$ or 2, may not refer to any service primitive $P$. 
where $Sap(P) \neq E$. Each protocol FSM has an initial state and a final state set. Zero or more protocol FSMs in an entity specification, called \textit{local protocol FSM}, are composed solely of service primitive transitions. Any other protocol FSMs in the entity specification, called \textit{synchronizing protocol FSMs}, should include send and receive transitions in addition to service primitive transitions. Send and receive transitions are labeled -$a$ and +$a$ to denote 'send message $a$' and 'receive message $a$', respectively. The set of messages that may be sent or received by different synchronizing protocol FSMs in an entity are disjoint with one another. The execution rule of a service primitive transition in an entity follows the first and second items of the definition of the reachability graph of a service specification. A send transition is executable once its head state$^a$ is reached; whereas the receive transition is executable only when the specified message has arrived at the concerned entity as the head element of the input queue, and when its head state is reached. It is possible for a protocol FSM to have more than one executable transitions at one time. If so, one of such transitions is nondeterministically selected to execute.

In the following section, we will present an algorithm to derive the protocol FSMs from given service FSMs: $M_1, \ldots, M_\alpha, N_1, \ldots, N_\beta$. We first assume that the underlying communication medium is free from transmission errors. This condition will

$^a$The state from which a transition is leaving is called the head state of the transition; the state to which the transition is entering is called the tail state of the transition.
be relaxed in Chapter V, where a transformation procedure is given to construct an error-recoverable protocol from its error-free version obtained through the protocol derivation algorithm.

4.2 From Service Specification to Protocol Specification

Each of the local constraint FSMs $M_1, \ldots, M_\alpha$ restricts the relative execution order of service primitives associated with a single SAP and thus can be embedded directly into entity specifications as the local protocol FSMs for Entity 1 or Entity 2 as decided by the associated SAP. On the other hand, global constraint FSMs $N_1, \ldots, N_\beta$ enforce the relative execution order of service primitives associated with different SAPs, and protocol entities serving different SAPs may communicate with each other only through message passing. Therefore, message exchanges are needed for synchronizing the service primitives associated with different SAPs so as to conform to the global constraint FSMs. So these messages are also known as synchronization messages. The algorithm to derive the synchronizing protocol FSM pair (two synchronizing protocol FSMs, one for Entity 1 and the other for Entity 2) from a global constraint FSM will be described in the following. Using the algorithm, we can derive a set of $\beta$ synchronizing protocol FSMs for each of Entity 1 and Entity 2 from the $\beta$ global constraint FSMs. Consequently, the $\beta$ synchronizing protocol FSMs for Entity
1 (Entity 2) together with the local protocol FSMs for Entity 1 (Entity 2) (i.e., the local constraint FSMs for Entity 1 (Entity 2)), directly coupled with one another, constitute the specification for Entity 1 (Entity 2).

It is likely that several different synchronizing protocol FSM pairs conform to a given global constraint FSM; therefore, in order to come up with a single synchronizing protocol FSM pair, the uncertainty should be removed somehow. First, we impose the same constraint, called decision constraint, as is done in [7]. The decision constraint says that (in terms of our context), w.r.t. a service state in a global constraint FSM, the decision as to which service primitive among the outgoing transitions associated with the same SAP will proceed next is made by the entity serving that SAP. Therefore, we require the protocol designer to give additional information, called decision option, concerning which entity should decide to let either itself execute next or the other entity to execute next. We identify the following three decision options, one of which needs to be assigned to each service state of a global constraint FSM. These decision options are the most possible and efficient decision patterns for synchronizing service primitives.

1. Decision is made by Entity 1, denoted as <1>.

2. Decision is made by Entity 2, denoted as <2>.

3. Decision is made by either Entity 1 or Entity 2, whichever has executed a service
primitive most recently, denoted as $<1|2>$.

The entity that makes the decision is called the decision entity of the corresponding service state; if a service state is assigned the decision option $<1|2>$, both Entity 1 and Entity 2 are the decision entities of it.

For service states having only incoming service primitive transitions associated with Entity 1 (Entity 2), its decision entity is Entity 1 (Entity 2) by default. This default assignment of decision options is to avoid redundant message transmissions. For example, in Fig. 4.3, the decision entity of State 2 in $N_1$ is Entity 1 since the only incoming service primitive transition is $Creq$, and $Sap(Creq) = SAP_1$.

Consider a global constraint FSM $N_i$ whose initial service state has outgoing service primitive transition $P_1$ associated with $SAP_1$ and $P_2$ associated with $SAP_2$. It is likely that $P_1$ and $P_2$ can be executed simultaneously (this condition can be checked by examining the reachability graph of the service FSMs as defined in Subsection 4.1.2), violating the constraint prescribed by $N_i$. Consequently, a measure has to be taken to resolve this collision problem. We choose to let the protocol designer assign to the initial state of $N_i$ an initiation option indicating which user has the right to initiate the operation of $N_i$. An initiation option should be specified explicitly along with the initial state pointer of any global constraint FSM encountering this problem. Including the initiation options as part of the service specification, we need to trim
the reachability graph of the service FSMs to get rid of the portion of the graph that is not supposed to be existent with these initiation options specified.

Our algorithm to derive the synchronizing protocol FSM pair from a given global constraint FSM, say $N_i$, is composed of three major steps as follows:

1. Insert required intermediate transitions between service primitive transitions in $N_i$ according to the specified decision option of each service state of $N_i$.

2. Adjust the initial state pointer according to the specified initiation option.

3. Project the resultant refined $N_i$ onto Entity 1 and Entity 2 independently to produce the desired synchronizing protocol FSM pair.

In the following subsections, we will describe each step in detail.

4.2.1 Step 1: Insertion of Intermediate Transitions

Step 1 is effectively a refinement of a global constraint FSM, say $N_i$, to incorporate the accompanying decision options. Nonconcurrent transitions and concurrent transitions are handled separately and we will proceed to handle nonconcurrent transitions first. For nonconcurrent transitions, the following substeps are taken:
For each service state (or node) of $N_i$ (a typical state $v$ is shown in Fig. 4.6(a)),

1. Partition the nonconcurrent transitions incident to state $v$ into four groups according to their associations (with the SAPs) and the directions (incoming or outgoing). The resulting partition $\{\{P_{11}, \ldots, P_{1m}\}, \{P_{21}, \ldots, P_{2n}\}, \{Q_{11}, \ldots, Q_{1r}\}, \{Q_{21}, \ldots, Q_{2s}\}\}$ for state $v$ is also shown in Fig. 4.6(a). The partition is due to the aforementioned decision constraint.

2. Create a small node for each group of transitions that is not empty. However, the small node for the group of incoming transitions associated with the decision entity is needed even if the group is empty. These small nodes are to be used for inserting intermediate transitions as described below. The case in which all $m, n, r, s$ are not zero is illustrated in Fig. 4.6(b).

3. If the decision option of state $v$ is $<1>$ (see Fig. 4.6(c)),

   (a) Create an edge, labeled by pair $(2 \rightarrow 1, a)$, from small node $w$ to small node $x$. The label indicates that Entity 2 should send Entity 1 synchronization message $a$.

   (b) Create an edge, labeled by pair $(1 \rightarrow 2, b)$, from small node $x$ to small node $y$. 
Figure 4.6. Illustration of Step 1 part for nonconcurrent transitions.
(c) Create an edge, labeled by $\varepsilon$, from small node $x$ to small node $z$.

Note that:

- The purpose that Entity 2 sends message $a$ to Entity 1 after the execution of one of $P_{2i}$, $i = 1, \ldots, n$ is to let Entity 1 decide which entity to execute next. If Entity 1 decides that Entity 2 executes next, it will send message $b$ to inform Entity 2 of the decision. If it decides to execute next itself, it goes instantaneously to state $z$ to select a service primitive for execution.

- To enable an entity to distinguish between the arrived messages resulting from different send transitions of the other entity, we require that each message in the labels of intermediate transitions have a distinct type (or name). Therefore, $a$ and $b$ are different synchronization messages and both are distinct from other messages that label intermediate transitions inside other service states.

- An intermediate transition need not be created if either of its small head or tail node is missing.

- The pair $(1 \rightarrow 2, a)$ is represented in the figures as

\[
\begin{array}{c}
1 \rightarrow 2 \\
\text{a}
\end{array}
\]
4. If the decision option of state $v$ is $<2>$ (see Fig. 4.6(d)), the required intermediate transitions can be similarly created.

5. If the decision option of state $v$ is $<1|2>$ (see Fig. 4.6(e)),

(a) Create an edge, labeled by pair $(1\rightarrow 2, a)$, from small node $x$ to small node $y$.

(b) Create an edge, labeled by $\epsilon$, from small node $x$ to small node $z$.

(c) Create an edge, labeled by pair $(2\rightarrow 1, b)$, from small node $w$ to small node $z$.

(d) Create an edge, labeled by $\epsilon$, from small node $w$ to small node $y$.

Note that:

- $a$ and $b$ are different synchronization messages and both are distinct from other messages that label intermediate transitions inside other service states.

- An intermediate transition need not be created if either of its small head or tail node is missing.

As for concurrent transitions, the way we make them work without losing synchronization is as follows: the entity that starts the concurrent transition sends a message to
the other entity immediately before it selects a service primitive to execute. The other entity also selects a service primitive to execute after receiving the same message. To make sure the ending of the concurrent transition, the entity that will make the decision for the next service primitive, say Entity A, should wait for the 'completion message' signifying the completion of a service primitive from the other entity after Entity A has completed its own service primitive. For each concurrent transition (a typical one tr is shown in Fig. 4.7(a)), the following substeps are taken:

1. This substep deals with state v, the head state of transition tr: create a small node w inside state v for tr and

   (a) If the decision option of state v is <1>, create an intermediate transition, labeled by pair (1→2, c), from small node x to small node w (see Fig. 4.7(b)).

   (b) If the decision option of state v is <2>, an intermediate transition is similarly created (see Fig. 4.7(c)).

   (c) If the decision option of node v is <1|2>, duplicate tr and small node w to yield tr' and small node w'; then create an edge, labeled by pair (1→2, c), from small node x to w and an edge, labeled by pair (2→1, d), from small node y to small node w' (see Fig. 4.7(d)). The duplication of tr and w is to make simple the transformation for concurrent transitions to be performed.
Figure 4.7. Illustration of Step 1 part for concurrent transitions.
in Step 3 (to be discussed in Subsection 4.2.3), since what we do here will maintain that there is only one intermediate transition preceding any concurrent transition.

Note that: \( c \) and \( d \) are different messages and both are distinct from any other message.

2. This substep deals with state \( u \), the tail state of transition \( tr \): create a small node \( z \) inside node \( u \) for \( tr \) and

(a) If the decision option of state \( u \) is \(<1>\), create an intermediate transition, labeled by pair \((2\rightarrow1, e)\), from small node \( z \) to small node \( x \) (see Fig. 4.7(e)).

(b) If the decision option of state \( u \) for \( tr \) is \(<2>\), an intermediate transition can be similarly created (see Fig. 4.7(f)).

Finally in Step 1, we mark as final all the small node(s) inside any final service state. Whether a communication service terminates or not depends on the execution history and future of service primitive transitions and should not be decided by either protocol entity. Thus, when the communication service terminates at a final service state, each protocol entity may be at any one of the small nodes inside the final service state. At the conclusion of Step 1, the service states of the global constraint FSM can be discarded and replaced by the small nodes and the inserted intermediate transitions.
4.2.2 Step 2: Adjustment of the Initial State Pointer

This step sets up the initial state pointer for the refined graph that incorporates the required intermediate transitions.

1. If the initial state has only outgoing transitions associated with SAP 1 (SAP 2), let the initial state pointer point to the small node created for this group of transitions. (see Fig. 4.8(a)-(b))

2. If the initial state has its outgoing transitions associated with both SAP 1 and SAP 2, let the initial state pointer point to the small node for the group of transitions that are outgoing from the initial service state and associated with the SAP specified in the initiation option. (see Fig. 4.8(c)-(d))

4.2.3 Step 3: Projection of the Refined FSM

To project the refined FSM obtained from Step 2 onto Entity 1, we first do some transformation for concurrent transitions: For each concurrent transition (a typical one $tr$ with label $[R_{11}, \ldots, R_{1t} \parallel R_{21}, \ldots, R_{2u}]$), along with its preceding and succeeding intermediate transitions,
Figure 4.8. Adjustment of the initial state pointer.
1. If the first components of the labels of its preceding and succeeding intermediate transitions are $1 \rightarrow 2$ and $1 \rightarrow 2$, respectively, perform the transformation illustrated in Fig. 4.9(a).

2. If the first components of the labels of its preceding and succeeding intermediate transitions are $1 \rightarrow 2$ and $2 \rightarrow 1$, respectively, perform the transformation illustrated in Fig. 4.9(b).

Note that message $b$ is called a completion message to signify the completion of the execution of a service primitive specified in the label of a concurrent transition. Due to the indefinite relative speed of both entities, Entity 1 may receive completion message $b$ before it begins executing its service primitive; therefore, a receive transition labeled $+b$ outgoing from state $w$ in Fig. 4.9(b) is needed to avoid the nonspecified reception error.

3. If the first components of the labels of its preceding and succeeding intermediate transitions are $2 \rightarrow 1$ and $1 \rightarrow 2$, respectively, perform the transformation illustrated in Fig. 4.9(c).

4. If the first components of the labels of its preceding and succeeding intermediate transitions are $2 \rightarrow 1$ and $2 \rightarrow 1$, respectively, perform the transformation illustrated in Fig. 4.9(d).

The note following substep (2) also applies here.
Figure 4.9. Transformation of a concurrent transition for deriving the protocol FSM for Entity 1.
After performing the transformation for concurrent transitions described above, we then change all the service primitive transitions associated with SAP 2 to $\varepsilon$ transitions; We also change the labels $(1\rightarrow2, a)$ and $(2\rightarrow1, a)$ to $-a$ and $+a$, respectively, for any message $a$. Finally we remove all the $\varepsilon$ transitions using the algorithm in [1]$^b$ to obtain a synchronizing protocol FSM for Entity 1. The synchronizing protocol FSM for Entity 2 can be derived similarly.

Applying the above algorithm to the examples in Fig. 4.2 and in Fig. 4.3, respectively, we obtain the protocol specifications shown in Fig. 4.10 and in Fig. 4.11,

---

The idea behind the algorithm is as follows:

i) If there exists an $\varepsilon$ cycle passing through a state, say $v$, of the FSM, the $\varepsilon$ cycle can be detected by constructing the reachability tree from state $v$. Since the number of states are finite, the reachability tree is finite and all the $\varepsilon$ cycles passing through $v$ can be detected. Doing this for every state of the FSM may detect all the $\varepsilon$ cycles in the FSM.

ii) An $\varepsilon$ cycle is removed by merging all the small nodes (along with their incident transitions) on the cycle.

iii) An $\varepsilon$ transition is removed by adding a copy of all the transitions outgoing from its tail state and then changing the head state of the copied transitions to be the head state of the $\varepsilon$ transition.

iv) Iterating Step iii) on each of the $\varepsilon$ transitions would ultimately end up with no $\varepsilon$ transitions in the FSM because no $\varepsilon$ cycles remain in the FSM from the first part of the algorithm.
respectively, where $M_1$ and $M_2$ are local protocol FSMs for Entity 1 and Entity 2, respectively, and the rest are synchronizing protocol FSMs for Entity 1 or Entity 2.
Figure 4.10. Derived protocol specification for the service specification given in Fig. 4.2.
Figure 4.11. Derived protocol specification for the service specification given in Fig. 4.3.
CHAPTER V

Error-Recovery Transformation

While the protocol derivation algorithm presented in Chapter IV can be applied to protocol layers whose underlying transmission service is reliable and FIFO, it is desirable to extend the algorithm to derive protocols with error-recovery capability such that protocol layers whose underlying transmission service may garble or lose messages can also be dealt with. The same kind of extension has been found in [24] for extending [52] and in [38] for extending [37]. We adopt an elaboration approach [38] to construct an error-recoverable (with error-prone medium assumption) two-entity protocol from its error-free version (with error-free medium assumption) produced from the protocol derivation algorithm. The procedure is best demonstrated through transformation rules. To achieve error-recovery, the popular error-recovery mechanism 'Positive Acknowledgment with Retransmission upon timeout (PAR)’ is chosen and embedded in the transformation rules.
This chapter is organized as follows. Section 5.1 first introduces the PAR mechanism and then presents the error-recovery transformation rules. Two problems would occur in a transformed protocol specification. Section 5.2 describes the problems and suggests remedies for getting rid of the problems. Finally, in Section 5.3, we compare our error-recovery treatment of communication protocols with related work.

5.1 Three Transformation Rules

The error-recovery transformation procedure consists of three transformation rules. Before presenting the transformation rules in detail in Sections 5.2 and 5.3, we describe the general behavior of the version of the PAR mechanism we use and the assumption of the underlying medium service in Section 5.1. Each transformation rule is composed of an original pattern of state transitions and a transformed pattern of state transitions. We will discuss the original patterns and transformed patterns in Sections 5.2 and 5.3, respectively.

5.1.1 PAR Mechanism

In the version of the PAR mechanism we use, the sender of a message, immediately after sending out the message, activates a timer for the message and sets a timeout
period for the timer. When a positive acknowledgment for the message is correctly received by the sender, the timer is disabled. When timeout occurs, a timeout event will trigger a state transition called timeout transition, denoted as $to$. We assume the timeout period is properly set so that the timeout occurs only if the message or its acknowledgment is garbled or lost, the so-called non-premature timeout. With the assumption of non-premature timeout, the acknowledgment for a message is not possible to arrive ungarbled after the timeout transition for the message has been executed. The receiver of a message or an acknowledgment will discard the message if it is detected to be garbled (provided that the lower layers contain a perfect error-detection mechanism) or if it is unexpected at the current state of the receiver. Therefore, message corruption is treated exactly the same way as message loss. For the sake of succinctness, the discarding of garbled and unexpected messages is implicit and not included as state transitions in an error-recoverable synchronizing protocol FSM. It is expected that the medium will eventually deliver a message correctly if the message is retransmitted a sufficient number of times; otherwise, with the permanent malfunction of the medium the PAR mechanism or any other mechanism is not possible to recover the message loss or corruption. This is referred as the fairness assumption of the communication medium.
5.1.2 Three Patterns of State Transitions

With respect to a send transition of a synchronizing protocol FSM produced from the protocol derivation algorithm, we may observe three distinct patterns of state transitions. In order to analyze how the three patterns are formed, we need to consider the purpose of one entity sending a message to the other. The purpose falls into four categories:

I) Decide to command the other entity to execute a service primitive.

II) Report the completion of a service primitive of a nonconcurrent transition and let the other entity decide whose service primitive to execute next.

III) Start a concurrent transition.

IV) Signify the completion of the execution of a service primitive specified in the label of a concurrent transition; the messages in this category are the so-called completion messages.

In the case that a send transition, say one labeled \(-a_i\), of a synchronizing protocol FSM is of Category I or II, the transition immediately succeeding the transition labeled \(-a_i\) must be a receive transition, enforced by the protocol derivation algorithm. The tail node of the send transition may be shared by the other send transitions, which is
caused by the substeps of removing ε cycles and ε transitions in Step 3 of the protocol derivation algorithm. Therefore the pattern shown in Fig. 5.1(a) is observed. For a send transition of Category III, two distinct patterns come up. The transition succeeding the service primitive transitions that result from the transformation of concurrent transition in Step 3 of the protocol derivation algorithm may be a receive or a send transition; the corresponding patterns are shown in Fig. 5.1(b) and Fig. 5.1(c), respectively. By the substep for concurrent transition in Step 1 of the protocol derivation algorithm, the decision entity of the service state succeeding a concurrent transition is always the entity other than the one sending the completion message. Therefore, the observed pattern w.r.t. a send transition of Category IV fits in that of Fig. 5.1(a). According to different patterns, different transformation rules are established to recover message loss and message corruption. In the following, we will first examine each pattern of state transitions separately, and then construct the transformed patterns to incorporate the PAR mechanism.

5.1.3 Transformed Patterns of State Transitions

Considering the first pattern in Fig. 5.1, after sending one of the messages $a_i, i = 1, \ldots, m$, the entity needs to remember the message just sent out in order to retransmit the right message. This requires the decoupling of the tail node of these send
Note:
The interested send transition is drawn in a fat line.

Figure 5.1. Patterns of state transitions w.r.t. a send transition.
transitions, namely node $w$, into $m$ different nodes. Any receive transition outgoing from node $w$ may serve as an acknowledgment of any of the messages $a_i, i = 1, \ldots, m$. Henceforth, a transformation rule transforming the pattern in Fig. 5.1(a) into the pattern in Fig. 5.2(a) can be established. The solid line portion of Fig. 5.2(a) is for the case in which node $w$ in Fig. 5.1(a) is not the initial state. If node $w$ happens to be the initial state, the dashed line portion should be added to form the resulting pattern; otherwise, no initial state can be identified in the transformed synchronizing protocol FSM.

For the second pattern in Fig. 5.1, the timeout event could happen not only in state $x$ but also in state $y$ since the execution of a service primitive transition is spontaneous; so the generated pattern of the second transformation rule is constructed as illustrated in Fig. 5.2(b).

As for the third pattern in Fig. 5.1, it suffices to recover the loss or corruption of message $b$, which is taken care of by the first transformation rule, and to disregard the loss or corruption of message $a$. This is simply because the two send transitions $-a$ and $-b$ are consecutive disregarding the possible intervening service primitive transitions and no other send or receive transitions may immediately succeed transition $-a$. This transformation rule leaves the synchronizing protocol FSM of the entity sending message $a$ intact, and only works on the 'mirror pattern', namely the corresponding
Figure 5.2. Patterns of state transitions after error-recovery transformation.
pattern in the corresponding protocol FSM of the other entity. The generated pattern for the other entity is shown in Fig. 5.2(c). In case that message $a$ is lost or garbled, message $b$ will arrive eventually due to the fairness assumption of the communication medium and that message $b$ would be retransmitted if lost or garbled according to the first transformation rule, justifying the existence of the receive transition $+b$ from state $x$ in the pattern of Fig. 5.2(c).

5.2 Sink-State and Duplicate Acceptance Problems

Applying the three transformation rules described in Section 5.2 to a synchronizing protocol FSM pair may cause two problems: sink-state problem and duplicate acceptance problem. In the following, we describe these two problems and suggest a remedy transformation for each of them.

5.2.1 Sink-State Problem

Consider a derived synchronizing protocol FSM pair in which one FSM has a sink state\(^a\) that is reachable via the interaction between protocol FSMs. After we apply

\(^a\)A state is called sink state if and only if there is no transitions outgoing from it.
the transformation rules to such a synchronizing protocol FSM pair, a portion of the transformed synchronizing protocol FSM pair would look like that shown in Fig. 5.3. Entity E enters a sink state \( v \) after executing a sequence of service primitive transitions (including \( \epsilon \) sequence) following the reception of one of messages \( a_1, \ldots, a_m \). The other entity, without knowing of the fact, will keep retransmitting that message forever without making any progress – a situation of *livelock*. Fortunately, the problem, we call *sink-state problem*, can be fixed by forcing Entity E to send a ‘sink command’ \( \sigma \) to the other entity once it reaches sink state \( v \) (see Fig. 5.4). Each sink command carries the identity of the synchronizing protocol FSM pair for which it is meant to work, so that messages for different synchronizing protocol FSM pairs can be distinguished. Notice that \( \sigma \) may be lost or garbled just like any other message, requiring the retransmission of it. The retransmission of \( \sigma \) can be triggered by the reception of any message that may be received prior to a sequence of service primitive transitions leading to state \( v \). In Figs. 5.3 and 5.4, \( a_1, \ldots, a_k \) are those messages that
Figure 5.4. Resulting patterns after fixing the sink-state problem.

may be received prior to a null sequence of service primitive transitions leading to state \( v \); and \( c \) stands for any message that may be received prior to a non-null sequence of service primitive transitions leading to state \( v \). In case that an entity reaches a sink state without previously participating in any communication with the other entity, by no means can the retransmission of \( \sigma \) be triggered. To deal with this difficulty, we may introduce two dummy messages exchanged between the two entities before the start of the synchronizing protocol FSM pair.

For a synchronizing protocol FSM derived from a consistent terminating communication service, a sink state in the FSM must be a final state. But, in the resulting patterns after fixing the sink-state problem as shown in Fig. 5.4, state \( v \) becomes non-final and state \( w \) becomes final; state \( u \) becomes non-final and state \( x \) becomes final.

A final state in a terminating protocol FSM may be not a sink state; so it is likely that the communicating service terminates when the current state of a synchronizing
protocol FSM in an entity is final but not a sink state. Then by no means can a sink command be sent out on behalf of the FSM by the entity and the other entity would keep retransmitting a message forever. The protocol entities simply can do nothing to take care of this since it is uncertain to protocol entities whether the service really terminates or a user will issue a service primitive later to continue the service.

5.2.2 Duplicate Acceptance Problem

A duplicate acceptance problem [45] may occur in a transformed synchronizing protocol FSM pair, thereby requiring a remedy. Fig. 5.5(a) shows the portion of the transformed synchronizing protocol FSM pair that faces the duplicate acceptance problem. If \( +a \) is not an outgoing transition from state \( w \), the received duplicate of message \( a \) will be discarded, according to our presumption about the behavior of the receiver of a message; therefore, both entities are still in synchronization. However, if \( +a \) is an outgoing transition from state \( w \), a situation of out of synchronization may be reached. The scenario of Fig. 5.5(b) shows such a situation. At state \( w \), after transition \( +a \) is executed, Entity 2 will be in state \( y \), thinking that message \( b \) has been correctly received by Entity 1, while Entity 1 is still in state \( x \), waiting for the arrival of message \( b \). Therefore, it requires a mechanism to discern between the message arrivals for the first \( +a \) transition, outgoing from state \( v' \), and that for the second one,
* stands for a sequence of service primitive transitions.

Figure 5.5. (a) Portion of a synchronizing protocol FSM pair facing the duplicate acceptance problem. (b) Space-time diagram for a scenario showing the occurrence of the duplicate acceptance problem.
outgoing from state $w$.

The usual solution to the duplicate acceptance problem is to attach a sequence number with each message transmission and to allocate each entity a sequence number variable for each message to determine whether the received message is duplicate or not. In any synchronizing protocol FSM pair produced from the protocol derivation algorithm and the transformation rules, whenever a message that could cause the duplicate acceptance problem is in transit, it is the only one on the channel; therefore, it is sufficient to use a single bit for the sequence number. For each message $a$ causing the duplicate acceptance problem (i.e., the 'a' in each occurrence of the pattern of Entity-2-part of Fig. 5.5(a) within the whole protocol FSM of Entities 1 and 2), two boolean (or bit) variables $f_a$ and $g_a$ are allocated to Entity 1 and Entity 2, respectively, and are initialized such that they have the initial values complementing to each other. The send transition $-a$ before any retransmission is modified to first complement the value of $f_a$ and then $-(a, f_a)$, meaning that the value of $f_a$ is attached to message $a$ for transmission. The send transition $-a$ for retransmission is modified to $-(a, f_a)$ (see Fig. 5.6(a)). The receive transition $+a$ is modified to $+(a, g_a)$, meaning that only received message $a$ attached with the value of $g_a$ will trigger this transition (see Fig. 5.6(b)). If the attached bit with message $a$ does not match the value of $g_a$, it is simply discarded according to our presumption about the behavior of a message receiver. It is worth noticing that if the rules in Fig. 5.6 are imposed on any message
Figure 5.6. Rules for fixing the duplicate acceptance problem.

in the protocol, the resulting protocol FSMs are even able to cope with an unreliable transmission medium service that may duplicate messages in addition to garbling and losing them.

In summary, the elaboration procedure for the error-recovery transformation consists of three transformation rules to incorporate the PAR mechanism and two posterior corrective operations to fix the two incurred problems. Applying the elaboration procedure to the synchronizing protocol FSMs shown in Fig. 4.10 and Fig. 4.11, respectively, results in the error-recoverable protocol FSMs illustrated in Fig. 5.7 and Fig. 5.8, respectively.
Rules for fixing duplicate acceptance problem should be applied to each message.

Figure 5.7. Error-recoverable protocol for two-slot bounded buffer service.
Figure 5.8. Error-recoverable protocol for the simplified ISO transport service.
5.3 Comparison with Related Work

There are two contributions made in [24] and [38] regarding the error-recovery extension on protocol synthesis. The approach of [24] is very general but primitive in that it only identifies the required receive transitions for garbled messages and timeout transitions in addition to correct messages. Therefore, to obtain the correct error-recovery operation, the protocol designer must specify the right send transitions and the right tail state of each receive and timeout transition. The approach of [38] does guarantee the correct error-recovery operation by enforcing the PAR mechanism on the protocol specifications. This is to trade the flexibility of error-recovery operation for the reduction in human intervention during the design process. We employ the elaboration approach proposed in [38]; therefore, the correct error-recovery operation is also guaranteed. However, to acknowledge a regular message received, we do not introduce the auxiliary acknowledgment messages as they do for each of their transformation rules. This is because we take advantage of the fixed patterns of state transitions w.r.t. each send transition in the protocol FSMs generated from our algorithm. Their work is in a general context, whereas ours is in a specialized context in which the service specification is restricted to our state-transition model.
CHAPTER VI

Protocol Analysis in the EFSM model

Since every protocol synthesis technique, including ours, has its own limitations imposed by the underlying model and assumptions, protocol analysis (validation and verification) is still needed to ensure the correctness of protocols beyond the expressive power of the underlying model for protocol synthesis techniques.

Protocol validation is an activity of detecting all possible syntactic errors in a given protocol specification. The syntactic errors of a protocol specification include communication deadlock, nonspecified reception, nonexecutable interaction, and improper termination. The definitions of these syntactical errors can be found in Section 3.1 of Chapter III. Owing to the complexity of the behavior of most communication protocols, protocol validation ought to be automated to avoid human mistakes in validating protocols. To make the automation of protocol validation possible, protocols to be validated have to be formally specified. A popular formal model for protocol speci-
fication is the Extended Finite State Machine (EFSM) model. The EFSM model has been used as the underlying model for many existing protocol specification languages, including Estelle [26] developed by the International Standards Organization (ISO) and the Enhanced Transmission Grammar (ETG) developed at The Ohio State University (see Appendix A for the syntax of the ETG protocol specification language).

Reachability analysis [50] has proved to be one of the most effective methods for protocol validation in state-transition models, including the EFSM model. A reachability analysis algorithm (program) constructs the global state graph that is a representation of all reachable global states and the immediate reachability relations between them. Nevertheless, protocol validation using reachability analysis often suffers from the global state explosion problem — the number of reachable global states explodes in such a way that the computer system does not have enough memory space to store the extremely huge global state graph and human user of the program (protocol designer) cannot tolerate the extensively long processing time. Approaches to relieving this problem have been the target of research in the area of protocol validation for many years [13, 14, 15, 18, 19, 20, 22, 25, 39, 48, 49, 51, 55]. In a previous research effort [25], we proposed a heuristics-based strategy that may help quickly locate syntactic errors of protocols specified in a Finite State Machine (FSM) model. In this regard, we present in this (second) part of research two global state graph reduction techniques that may alleviate the global state explosion problem for
protocol validation in the Extended Finite State Machine (EFSM) model, which is an extension of the FSM model.

In this chapter we consider the Extended Finite State Machine (EFSM) model for protocol specification and the use of reachability analysis to validate protocol specifications in the EFSM model. In Section 6.1, we introduce the EFSM model. In Section 6.2, the use of reachability analysis to perform protocol validation is described.

6.1 The EFSM Model

The evolution of the EFSM model may be traced back to the work in [6]. The version of the EFSM model considered in this dissertation is one of the most widely accepted models for specifying communication protocols. In this version of the EFSM model, the behavior of each protocol entity is described as an Finite State Machine (FSM) and a set of context variables declared for the entity that may be accessed during state transitions. The FSMs in a protocol specification are disjoint with one another, so are the sets of context variables. Protocol entities may communicate with one another only via sending and receiving messages through a number of error-free First-In-First-Out (FIFO) unidirectional queues (channels). We assume that, between each pair of entities, there exists at most two channels, in opposite directions to each other. This assumption, however, does not restrict the applicable scope of the reduction
Each state transition is associated with a label consisting of two parts: a condition followed by an action. Fig. 6.1 shows the generic state transition along with the associated label in the state-transition diagram representation of the EFSM model. A horizontal bar is employed to separate the condition and the action parts of the label. A condition is a sequence of predicates on context variables and/or receive events. An action is a sequence of operations on context variables and/or send events. For example, "?D(x); x > 0" is a condition, where '?'D(x)’ denotes a receive event that, upon the arrival of a message of type D at the input queue as the head element, receives the message and the appended value of the message is assigned to the context variable x; and 'x > 0' denotes a predicate on x that tests whether the value taken on by x is greater than zero or not. On the other hand, "y := x + 1; !A(y)" is an example
of an action, where \( y := x + 1 \) denotes an operation on \( y \) that assigns the value of the expression \( x + 1 \) to \( y \); and \( !A(y) \) is a send event that sends a message of type \( A \) and appends the value taken on by \( y \) to the message. We borrow the notation of Communicating Sequential Processes (CSP) [17] in using '?' and '!' to stand for the send and receive events, respectively. In our context, '?' is a blocking receive as is in CSP whereas '!' is a non-blocking send that will not wait for the communication partner to be ready for the send event to be executable, as is required for the '!' in CSP.

A transition in the FSM describing an entity is executable when the entity has reached the head state\(^a\) of the transition and the condition part of the transition becomes TRUE. A condition becomes TRUE if and only if all its component predicates and receive events become TRUE, when evaluated in the specified order. A null condition is vacuously TRUE at any point of time. A receive event becomes TRUE if a message of the specified message type has arrived at the entity as the head element of the specified channel. The evaluation of a receive event has the side effect of dequeuing that message and assigning the associated parameter values to the specified context variables. The evaluation of a predicate, however, is assumed to have no side effects on context variables. Executing an action results in executing the component operations.

\(^a\)The state from (to) which a transition is leaving (entering) is called the head state (tail state) of the transition.
and/or send events in the specified order.

The operations on context variables are normally used to change the values taken on by some context variables and the predicates to check the truth of the associated boolean expressions on context variables. A send or receive event is accompanied by a parameter list, possibly a null list. The parameter list accompanying a send event is a list of expressions on context variables, whereas the parameter list accompanying a receive event is a list of context variables. The values taken on by the expressions in the parameter list accompanying a send event are appended to the message, in the specified order, immediately before being sent out at the occurrence of the send event. Immediately after a message is received at the occurrence of a receive event, the appended values of the message are assigned to the context variables in the parameter list accompanying the receive event, in the specified order.

It is quite possible that more than one transition in an FSM are executable at a certain time. In this case, exactly one of such transitions is nondeterministically chosen to execute. Transitions are atomic in the sense that the side effect of evaluating the condition of a transition should be undone if the transition turns out to be not executable or executable but not chosen for execution; and the whole effect of executing a chosen transition is exhibited instantaneously and no intermediate outcome is visible.

A context variable is accessed in two ways: reference and assignment. A context
variable is said to be referenced (or read) when it is accessed in a predicate, in the right-hand side of an infix assignment operator (e.g., ‘:=' operator in Pascal), or in the parameter list of a send event. A context variable is said to be assigned (or written) when it is accessed in the left-hand side of an infix assignment operator or in the parameter list of a receive event. A context variable may be initialized, thus defined initially and forever; otherwise, it is undefined initially and becomes defined from the moment when it is first assigned.

As an example shown in Fig. 6.2, a version of the Alternating Bit Protocol (ABP) [2] is specified as four FSMs connected in a ring. In this example, the communication media may only garble a message into a detectable error message or mutate a piggyback bit 0 (1) into 1 (0) in addition to correctly delivering messages. In this protocol an alternating bit variable is kept by both the sender and the receiver, and each message sent is appended with a bit value. With this setting, the sender is able to retransmit a message if the message or its corresponding acknowledgment is garbled or the appended bit is mutated; and the receiver is able to determine whether a message received is a duplicate or not.
Figure 6.2. Specification of ABP in the EFSM model.
6.2 Protocol Validation using Reachability Analysis

In the EFSM model, a global state may be represented as a matrix (called *global state matrix*) shown in Fig. 6.3, where each diagonal element records the state of the corresponding entity and the values taken on by the context variables declared for the entity; and each off-diagonal element records the sequence of messages, possibly with appended parameter values, in transit on the corresponding queue.

Let $S$ be a typical global state that is worked on by a reachability algorithm. To generate another global state $S'$ from $S$, a state transition of some entity, say Entity $i$, that is executable at $S$ is executed. Global state $S'$ is identical to global state $S$ except possibly the state of Entity $i$, the values taken on by context variables for Entity $i$, and the contents of some input and/or output channels of Entity $i$. We then say that global state $S'$ is *immediately reachable* from global state $S$. A global state, say $S''$ is *reachable* from global state $S$ if and only if $S''$ is immediately reachable from $S$ or there exists a sequence of global states, say $S_1, S_2, \ldots, S_n$, where $n \geq 1$, such that $S_{i+1}$ is immediately reachable from $S_i$, for $i = 1, \ldots, n - 1$, $S_1$ is immediately reachable from $S$, and $S''$ is immediately reachable from $S_n$.

The initial global state in the EFSM model is defined as a global state matrix whose diagonal elements are the initial state of the corresponding FSM and initial
\( v_i \): the state of Entity \( i \), \( i = 1, \ldots, N \)

\( n_i \): the number of context variables declared for Entity \( i \), \( i = 1, \ldots, N \)

\( x_{ik} \): the \( k \)th context variable of Entity \( i \), 
\( i = 1, \ldots, N \), \( k = 1, \ldots, n_i \)

\( a_{ik} \): the value taken on by \( x_{ik} \)

\( c_{ij} \): the sequence of messages, with appended parameter values, in transit on the channel from Entity \( i \) to Entity \( j \), \( i, j = 1, \ldots, N \), \( i \neq j \)

Figure 6.3. Global state representation in the EFSM model.
values taken on by context variables declared for the corresponding entity; and whose off-diagonal elements are empty sequences. The reachability algorithm constructs the global state graph recording all global states reachable from the initial global state and the immediate reachability relation between these global states.

Syntactic errors of protocols can be defined as properties of the global state graph; thus, by checking the global state graph, we can detect the existence of any syntactic errors of protocols. Syntactic errors of protocols can be checked either in parallel with the generation of the global state graph or after the whole global state graph is constructed. Consequently, as can be seen from the above discussion, protocol validation using reachability analysis can be easily automated once the global state graph can be formally defined according to the protocol specification.
CHAPTER VII

Global State Graph Reduction Techniques

As mentioned in Chapter VI, protocol validation using reachability analysis often suffers from the global state explosion problem so that reachability analysis is only suitable to be applied to protocols whose global state graph is not huge. Fortunately, the problem can be alleviated by introducing global state graph reduction techniques into reachability analysis. In this chapter, we present two such reduction techniques that may relieve the global state explosion problem for protocol validation in the Extended Finite State Machine (EFSM) model.

The chapter is organized as follows. In Section 7.1, an observation is given so as to motivate the derivation of the first global state graph reduction technique. Also provided in Section 7.1 is an upper bound for the effect of the technique on the reduction of the global state graph. In Section 7.2, the impact of the reduction
technique on the incremental protocol validation is examined. Finally, the second
reduction-technique based on a similar reasoning as the first one is presented in Section
7.3.

7.1 The First Reduction Technique

In this section, we first present a global state graph reduction technique for the
EFSM model, and then provide an upper bound of the reduction ratio, which is a
measure of the percentage of the number of global states that can be reduced using
this technique.

7.1.1 The Technique

This global state graph reduction technique requires the data flow analysis on all
FSMs describing protocol entities to obtain the information called dead variable sets.
Based on these dead variable sets a global state equivalence may be defined so that
only one global state in an equivalence class needs to be generated during the global
state graph generation (see Fig. 7.1). In the following, we will explain what the dead
variables are, how to use dead variable sets to define the global state equivalence for
reducing the global state graph, and how to obtain dead variable sets.
Let $x$ be a context variable declared for Entity $e$ and $v$ be a state of the FSM describing Entity $e$. Then $x$ is said to be dead at $v$ if and only if, starting from State $v$, $x$ will not be referenced in all possible future execution paths of Entity $e$ or the next possible accesses to $x$ in all possible future execution paths of Entity $e$ are assignments. For instance, in the ABP specification shown in Fig. 6.2, variable $y$ is dead at every state of Entity Sender. This is because $y$ is referenced in two transitions, one from State 3 to State 1 and the other from State 3 to State 2; and in the label of either transition the reference to $y$ is immediately preceded by a receive event assigning a value to $y$; thus, starting from every state of Entity Sender, all possible next accesses to $y$ are assignments. By the same argument, $y''$ is dead at every state of Entity Receiver.

The future behavior of a protocol at a global state can be defined as the set of all possible future execution paths of the protocol starting from the global state. The
future behavior of a protocol entity at an entity state is defined as the set of all possible future execution paths of the entity starting from the entity state. For the purpose of protocol validation, it is sufficient to capture all reachable global states and all the immediate reachability transitions among them. Therefore, two global states may be considered as equivalent if their future behaviors are identical. This consideration of global state equivalence is virtually the same as that defined in [5].

It is observed that the future behavior of Entity e at State v does not depend on the current value taken on by a variable x that is dead at v since, starting from State v, either x will not be referenced or x will be referenced only after an assignment to x, which will destroy the current value taken on by x at State v prior to any reference to x. Notice that this observation holds no matter what the current value of any other variable is, what the current content of any message queue is, and what any other entity will behave in the future. Consequently, we may safely consider as equivalent any two global state matrices GS1 and GS2 that differ from each other only in the values taken on by context variables that are dead at either of the two global states. For example, the two global state matrices shown in Fig. 7.2 for the ABP specification in Fig. 6.2 are considered equivalent since context variables y and y'' are dead at State

\begin{flushright}
\textit{A context variable declared for Entity i is dead at global state GS if and only if it is dead at the state of Entity i in GS. According to the definition of this global state equivalence, the states of each entity in GS1 and GS2 must be identical to each other. As a result, a context variable is dead at GS1 if and only if it is also dead at GS2.}
\end{flushright}
1 of Entity Sender and State 1 of Entity Receiver, respectively. Given this global state equivalence, it is sufficient for a global state representation to record only the values taken on by context variables that are live at a global state. To achieve the same effect, we may force a context variable that is dead at a global state to take on the special constant value ‘d’, meaning that the actual value taken on by this context variable is not needed for performing the global state matrix matching in the process of global state graph generation. Below we show that the global state equivalence defined above indeed guarantees the identity of future behaviors of equivalent global states.

**Theorem 1 (Reduction Technique 1)** Let $SS$ be a set of global state matrices that differ from one another only in the values taken on by context variables that are dead at the global state matrices in $SS$. Then the future behaviors of the global state matrices in $SS$ are all identical.

**Proof:** We wish to show that any execution path, say $p$, starting from an arbitrary global state matrix, say $S_i$, in $SS$ must be also executable starting from any other global state matrix, say $S_j$, in $SS$. Once the statement is established, we can assure that the sets of all possible future execution paths starting from the global state matrices in $SS$ are all identical.

We start projecting path $p$ onto any entity, say Entity $k$, to obtain an execution
Figure 7.2. Two equivalent global states for ABP by the first reduction technique.
Let $x_1, \ldots, x_n$ be the context variables declared for Entity $k$ that are dead in $S_i$ (or in $S_j$ equivalently, as $S_i$ and $S_j$ must have the same values in the corresponding entity state elements according to the definition of $SS$). Any of $x_1, \ldots, x_n$ will not be referenced until it is assigned along path $pk$ according to the definition of dead variables. Therefore, the values taken on by any of $x_1, \ldots, x_n$ in $S_i$ and that in $S_j$ do not affect the executability of path $pk$. They do not affect the executability of path $p$ either, for no entities other than Entity $k$ can reference any of $x_1, \ldots, x_n$. The statement is true for any other context variable that is dead at $S_i$ (and at $S_j$). Furthermore, $S_i$ and $S_j$ are different from each other only in the values taken on by such context variables. Therefore, path $p$ should be also executable starting from $S_j$.

The problem of deciding whether a context variable is dead at a state of the FSM describing an entity is identical to the problem associated with live variable analysis [31], which decides whether a variable is live (or not dead) at a certain point in a sequential program. With the definition of dead variable given at the beginning of this section, we say that context variable $x$ is live at State $v$ of Entity $e$ if and only if, starting from State $v$, there exists an execution path of Entity $e$ in which the first access to $x$ is a reference. We now formulate the data flow equation for live variable analysis in terms of the EFSM model by modifying the one given in [31]. Let
$LIVE(v)$: the set of context variables which are live at State $v$

$FREF(v, u)$: the set of context variables to which the first accesses in the label of the transition from State $v$ to State $u$ are references

$ASGN(v, u)$: the set of context variables which are assigned in the label of the transition from State $v$ to State $u$

The data flow equation that relates the $LIVE$ set of State $v$ to the $LIVE$ sets of the successor states of State $v^b$ is

$$LIVE(v) = \bigcup_{u \in S(v)} FREF(v, u) \bigcup_{u \in S(v)} (LIVE(u) - ASGN(v, u)), \quad (7.1)$$

where $S(v)$ is the set of successor states of state $v$. The explanation for the term $\bigcup_{u \in S(v)} FREF(v, u)$ in Equation 7.1 is as follows: if a context variable, say $x$, is accessed in the label of a transition from State $v$ to a state, say State $u$, and the first access to $x$ in the label of the transition is a reference; then $x$ is live at State $v$ immediately following the definition of live variables. The explanation for the term $\bigcup_{u \in S(v)} (LIVE(u) - ASGN(v, u))$ in Equation 7.1 is as follows: if a context variable, say $y$, is live at a successor state of State $v$, say State $u$, and it is not assigned in the label of the transition from State $v$ to State $u$; then $y$ should be live at State $v$, too.

Since State $v$ can be any state of the FSM, a system of $n$ data flow equations can be formed for an FSM of $n$ states. The live variable analysis algorithm takes $O(n^2)$.

---

$^b$ $u$ is a successor state of state $v$ if and only if there is a state transition from $v$ to $u$. 
time for solving an arbitrary system of $n$ such data flow equations. Interested readers are referred to [31] for further details on various live variable analysis algorithms for solving systems of $n$ such data flow equations either for arbitrary FSMs or for some restrictive class of FSMs.

It is worthwhile noting that the problem associated with live variable analysis can also be formulated in terms of the branching-time temporal logic CTL (Computation Tree Logic) [10]. Consequently, the model checking algorithm proposed in [10] may be used to find whether a variable is live at a certain point of a sequential program. Let $G$ be the control flow graph of the program. The model checking algorithm will take time $O(n + d)$, where $n$ and $d$ are the number of nodes and the number of edges in $G$, respectively. More details about applying the model checking algorithm to the live variable analysis can be found in Appendix C.

Having obtained the $LIVE$ sets for all states of every FSM, we simply complement the sets to get the corresponding dead variable sets and use the above-mentioned global state equivalence to generate the global state graph in the second stage of the reduction technique.

In the following, we show an upper bound of a measure indicating the performance of this reduction technique. This upper bound is parameterized by the number of possible values taken on by each context variable. Although this upper bound is
very rough, it provides some idea of how the technique may best perform on a given protocol specification.

7.1.2 An Upper Bound of Reduction Ratio

Let $GR$ and $G$ be the global state graphs generated with and without the reduction technique, respectively; and $\tau$ and $T$ be the total number of global states in $GR$ and $G$, respectively. Then the reduction ratio $\eta$, i.e., the percentage of reachable global states saved, can be defined as

$$\eta = \frac{T - \tau}{T} = 1 - \frac{\tau}{T}.$$  

In general, the value of $\eta$ depends heavily on how the FSMs interact with one another, which in turn depends on how the references and assignments to the context variables are distributed among the state transitions and also on the number of possible values taken on by each context variable. However, the greatest lower bound and an upper bound for $\eta$ can be easily obtained as follows. The greatest lower bound is zero since $\eta$ is a nonnegative measure and this lower bound is met by a protocol specification that does not contain any context variable that is dead at some entity state. To find an appropriate upper bound, we first define several parameters that may effect the values of $\eta$. Let
N: the number of protocol entities

Vi: the set of states in Entity i, i = 1, . . . , N

DEAD(v): the set of context variables that are dead at State v, i.e., the complementary set of LIVE(v)

r(x): the number of possible values including the undefined value taken on by context variable x, i.e., the range of x

Consider the r global states in GR, each of which is assigned a distinct number from 1 to r. Suppose that there are Ik global states in G that are equivalent to global state k in GR. It is clear that, for any k = 1, . . . , r,

$$l_k \leq \prod_{i=1}^{N} (\max_{v \in V_i} \prod_{x \in \text{DEAD}(v)} r(x))).$$

Then

$$T = \sum_{k=1}^{r} l_k \leq r \prod_{i=1}^{N} (\max_{v \in V_i} \prod_{x \in \text{DEAD}(v)} r(x))).$$

Thus

$$\eta = 1 - \frac{T}{T} \leq 1 - \frac{r}{r \prod_{i=1}^{N} (\max_{v \in V_i} (\prod_{x \in \text{DEAD}(v)} r(x)))} \leq 1 - \frac{1}{\prod_{i=1}^{N} (\max_{v \in V_i} (\prod_{x \in \text{DEAD}(v)} r(x)))}$$

From this upper bound, we expect that a protocol specification with context variables of wider ranges may have a higher reduction ratio than a similar protocol specification.
with context variables of narrower ranges. For the ABP protocol specification as shown in Fig. 6.2, the upper bound of the reduction ratio is 80/81.

Having fewer global states and fewer global state transitions in the global state graph means less space is required to store the global state graph and less time is consumed to generate the global state graph. The tradeoff between the time/space used for live variable analysis and that saved for global state graph generation depends on the reduction ratio and the ratio of the size of the global state graph vs. the size of each FSM. The reduction ratio, however, is very hard to be estimated from a given protocol specification without actually generating the global state graph of the protocol specification. As for the ratio of the size of the global state graph vs. the size of each FSM, according to [18], the global state graphs of most real-life protocols are much larger than each FSM. Consequently, we may expect that the incorporation of a live variable analysis algorithm may be worthwhile even with a very low reduction ratio since the incurred overhead may be negligible compared to the time/space used for generating the global state graph.

To further support the above point, we compare the worst-case time complexity of the live variable analysis algorithm and the global state graph generation algorithm. The incurred overhead of the live variable analysis is $O(Nn^2)$ for an N-entity protocol with $n$ states in each FSM. The global state graph generation algorithm would take
time proportional to the size of the global state graph. The maximal possible number of global states in the global state graph can be figured out from some parameters as follows. For simplicity, we consider a particular protocol specification with the following parameters.

\( b \): the capacity of each channel

\( k \): the number of different messages that may flow through each channel

\( c \): the number of context variables declared for each entity

\( r \): the number of possible values including the undefined value taken on by each context variable

The maximal possible number of global states is

\[
(nr)^N (1 + k + k^2 + \cdots + k^N)^{N-1} = \frac{r^{b+1} - 1}{k-1} N^{N-1}.
\]

From the above results it is obvious that the worst-case time complexity of the global state graph generation algorithm is much larger than that of the live variable analysis algorithm.

The reduction technique has been incorporated into our protocol validation system for the ETG language, which runs under OSx on a Pyramid machine. Running the system for the ETG specification of the ABP shown in Fig. 6.2 (see Appendix B for the
output of running the system on ABP), only 56 global states were generated, instead of 158 global states that would be generated without using the reduction technique. This is a reduction of about 2/3 in the number of global states generated.

7.2 Impact of the First Reduction Technique on Incremental Protocol Validation

For a protocol with a large global state graph, it is inefficient and unnecessary for a protocol validation system to reconstruct the new global state graph after a change in the protocol specification. An incremental update algorithm is thus needed to modify the global state graph constructed previously to reflect the change. An "incrementalized" protocol validation system then consists of two phases: the global state graph generation phase and the global state graph update phase; the former is passed through just once but the latter maybe several times (see Fig. 7.3). Suppose the global state graph reduction technique given in the last section is incorporated in the global state graph generation phase. Then in the global state graph update phase, a change in the specification of an entity in a protocol may entail the change of the \textit{LIVE} sets for some states of the FSM describing the entity. An incremental data flow analysis algorithm such as those in [40, 41] can be used to figure out the new
The global state graph may also change due to the change in the specification of the entity. Here we consider three types of changes which are localized with respect to the whole structure of the FSM describing an entity.

1. addition of a state transition to an FSM

2. deletion of a state transition from an FSM

3. change of a state transition label in an FSM

The first two types of changes are structural whereas the third one is non-structural since the first two modify the structure of the FSM while the third does not. The
required updates on the global state graph resulting from each type of changes are discussed in the following by the listed order.

### 7.2.1 Addition of a State Transition

By examining the data flow equation given in Equation (7.1), we find that the addition of a state transition to an FSM would enlarge the \(LIVE\) sets (or equivalently reduce the \(DEAD\) sets) for some states of the FSM, making inequivalent some global states that are equivalent originally. The global state graph may be updated first by decoupling each global state to form a set of global states that are considered inequivalent under the new \(DEAD\) sets and by decoupling the incident global state transitions accordingly; and then by adding all global states and global state transitions that are reachable only via sequences of global state transitions that contain a global state transition caused by the added state transition of the entity. In order to accurately decouple the global states at this phase, some bookkeeping routine has to be included during the global state graph generation phase of the validation system. A possible bookkeeping routine is given in the next paragraph.

Let \(S\) be a typical global state that is worked on by an incrementalized protocol validation system at the global state graph generation phase. To generate another
global state $S'$ from $S$, a state transition of some entity, say Entity $i$, that is executable at $S$ is executed. Global state $S'$ is identical to global state $S$ except possibly the state of Entity $i$, the values taken on by context variables for Entity $i$, and the contents of some input and/or output channels of Entity $i$. The bookkeeping routine records, along with the global state transition, the values taken on by the context variables that are dead at $S'$. The values taken on by the context variables that are live at $S'$ are already recorded in $S'$ and need not be recorded along with the global state transition to avoid double work. Since a global state transition may occur many times, the information recorded along with the global state transition should be actually a set \( \{<x_1=a_1, \ldots, x_n=a_n>|x_1, \ldots, x_n \text{ are dead at } S'\} \), called snapshot set, where each tuple in the set represents a possible snapshot of the context variables that are dead at $S'$ upon the completion of the execution of the global state transition. If the range of each context variable is finite as is true for the context variables used in any real-life protocols, the number of possible snapshots recorded along with any global state transition in the global state graph is also finite. In order to completely capture all possible snapshots, when a global state equivalent to a previously generated global state is generated, the newly added snapshot for a global state transition incoming to the global state has to be propagated until no new snapshot information can be added to the snapshot set for any global state transition in the global state graph.

Now we consider how the decoupling of global states at the global state graph
update phase can be done, given the global state graph enriched with the snapshot sets recorded along with each global state transition in the global state graph. Let the snapshot set $\text{snap}$ along with a global state transition incoming to global state $S'$ be $\{ <x_1 = a_1, \ldots, x_n = a_n> | x_1, \ldots, x_n \text{ are dead at } S' \}$. Suppose after the addition of a state transition to an FSM, context variables $x_{j_1}, \ldots, x_{j_k}$ become live at $S'$, where $\{x_{j_1}, \ldots, x_{j_k}\} \subseteq \{x_1, \ldots, x_n\}$. For each maximal subset of $\text{snap}$ with fixed values, say $a_{j_1}, \ldots, a_{j_k}$, taken on by $x_{j_1}, \ldots, x_{j_k}$, respectively, we create a global state $S''$ that is derived from $S'$ by changing the values taken on by $x_{j_1}, \ldots, x_{j_k}$ from 'd', $\ldots$, 'd' to $a_{j_1}, \ldots, a_{j_k}$, respectively. A global state transition incoming to $S''$ is then created with the snapshot set $s_n$, where $s_n$ is the projection of the maximal subset of $\text{snap}$ with $x_{j_1} = a_{j_1}, \ldots, x_{j_k} = a_{j_k}$ onto $\{x_1, \ldots, x_n\} - \{x_{j_1}, \ldots, x_{j_k}\}$, i.e.,

$$s_n = \pi_{\{x_1, \ldots, x_n\} - \{x_{j_1}, \ldots, x_{j_k}\}}(\sigma_{x_{j_1} = a_{j_1}, \ldots, x_{j_k} = a_{j_k}}\text{snap}),$$

where $\sigma$ and $\pi$ are selection and projection operators in relational algebra [46]. Note that it is possible that some equivalent global states are created by the decoupling procedure; therefore, a matching with previously created global states is needed before a new one can be created.

One disadvantage of the bookkeeping routine is obviously that a great amount of time/space is consumed when the global state graph is large. However, compared

\*\*A subset $\text{SUB}$ of a set $\text{SET}$ with some property is said to be maximal if and only if no other subsets of $\text{SET}$ with the same property properly contain $\text{SUB}$.
to the incrementalized protocol validation system without incorporating the reduction technique, the incrementalized protocol validation system incorporating the reduction technique still saves a lot of space/time at the global state generation phase since the generated global state graph is smaller although more information needs to be recorded along with global state transitions. As for the tradeoff between the time spent on the data flow analysis, the bookkeeping routine, the decoupling procedure of the incrementalized protocol validation system with the reduction technique, and the extra time spent on global state graph generation of the incrementalized protocol validation system without the reduction technique, it is very dependent on the original protocol specification and the follow-up changes. Therefore, it is very hard to decide whether the investment of the reduction technique in an incrementalized protocol validation system is worthwhile or not.

In the bookkeeping routine, it may not be necessary to record all the snapshots along with global state transitions due to the fact that some context variables being dead at some global states all the time during the global state graph update phase. However, which context variables are dead at which global states all the time is not predictable at the global state graph generation phase. As a result, to avoid recording redundant snapshots, the bookkeeping routine has to be moved to the global state graph update phase, recording only the required snapshots for making correct decoupling after each addition of a state transition.
7.2.2 Deletion of a State Transition

The deletion of a state transition from an FSM would reduce the \textit{LIVE} sets (or equivalently enlarge the \textit{DEAD} sets) for some states of the FSM, making equivalent some global states that are not equivalent originally. Therefore, the global state graph may be updated first by removing all global states and global state transitions that are reachable \textit{only} via sequences of global state transitions that contain a global state transition caused by the deleted state transition of the entity; and then by merging each set of global states that are equivalent under the new \textit{DEAD} sets into one global state and merging the incident global state transitions accordingly.

Note that merging each set of global states that are equivalent under the new \textit{DEAD} sets could take an extensive amount of time. Besides, part of the merging could be undone by the decoupling procedure for an addition of a state transition to an FSM that is performed later than this deletion of a state transition from an FSM. It may be desirable to leave alone the sets of equivalent global states without merging.
7.2.3 Change of a Transition Label

Changing the label $A$ of a state transition, say $t$, to the label $B$ has the same effect as deleting the state transition $t$ with label $A$ and then adding the state transition $t$ with label $B$. Therefore, the update of the global state graph to reflect this type of changes can be done using the global state graph update procedures for the other two types of changes. However, it seems that there is a more efficient way to do this, which is the subject for further study.

7.3 The Second Reduction Technique

The second reduction technique that has been incorporated into our protocol validation system is based on a similar reasoning as is the first reduction technique discussed in Section 7.1. However, this second reduction technique works on the assumption that no context variables would be referenced when they are undefined. Referencing any undefined context variables appears to be meaningless. Moreover, any such attempts can be detected by the validation system and then be corrected by the protocol designer. So the above assumption may be reasonably valid. With this assumption, starting from a global state, an undefined context variable will not be referenced in the future until it is assigned and becomes defined thereafter.
Consider two global state matrices of a protocol specification, $S_1$ and $S_2$, with identical corresponding element values except the value taken on by a context variable $x$, which is undefined in $S_1$ and defined in $S_2$. In all possible future execution paths of the protocol starting from $S_1$, $x$ will not be referenced or the next possible accesses to $x$ are assignments according to the aforementioned assumption. Therefore, simply assigning an arbitrary value to $x$ in $S_1$ does not change the possible future execution paths of the protocol starting from $S_1$, thereby leading to the consideration of the equivalence between $S_1$ and $S_2$. By the transitivity of an equivalence relation, we may consider as equivalent any two global state matrices that differ from each other only in the values taken on by the context variables that are undefined in one of the two global state matrices. For example, the two global state matrices shown in Fig. 7.4 for the ABP specification in Fig. 6.2 are considered equivalent since they differ from each other only in the values taken on by context variables $y$ and $y''$, which are undefined in one of the two global state matrices.

The global state equivalence can be further generalized as follows: a set of global state matrices form an equivalence class if they differ from one another only in the values taken on by the context variables that are undefined in one the global state matrices. For example, assume that three global state matrices, $S_1, S_2$, and $S_3$, are identical to one another except the values taken on by context variables $w, x$, and $y$. In $S_1, S_2$, and $S_3$, the values taken on by $w, x$, and $y$ are $<w = ?; x = 1; y = 2>$,
Figure 7.4. Two equivalent global states for ABP by the second reduction technique.
<w = 3; x = ?; y = 4>, and <w = 5; x = 6; y = ?>, respectively, where '?' denotes the undefined value. The three global state matrices are equivalent according to the generalized definition of global state equivalence, but not according to the definition of global state equivalence discussed in the previous paragraph. Below we show that the global state equivalence defined in this paragraph indeed guarantees the identicalness of future behaviors of equivalent global states.

**Theorem 2 (Reduction Technique 2)** Assume that no context variables would be referenced when they are undefined (called basic assumption hereafter). Let SS be a set of global state matrices that differ from one another only in the values taken on by the context variables that are undefined in one of the global state matrices in SS. Then the future behaviors of the global state matrices in SS are all identical.

**Proof:** We wish to show that any execution path, say \( p \), starting from an arbitrary global state matrix, say \( S_i \), in SS must be also executable starting from any other global state matrix, say \( S_j \), in SS. Once the statement is established, we can assure that the sets of all possible future execution paths starting from the global states in SS are all identical.

Suppose that, in \( S_i \), context variables \( u_1, \ldots, u_a \) are undefined; context variables \( x_1, \ldots, x_b \) are defined and take on the same values as those taken on by the corresponding context variables in \( S_j \); context variables \( y_1, \ldots, y_c \) are defined
and take on different values as those taken on by the corresponding context variables in $S_j$. Along the execution path $p$, any of $u_1, \ldots, u_a$ would be referenced only after it has been assigned due to the basic assumption. Thus values taken on by $u_1, \ldots, u_a$ in $S_j$ has no effect on the executability of path $p$ starting from $S_j$.

In the next paragraph, we will show by contradiction that any of $y_1, \ldots, y_e$ will be referenced only after being assigned along path $p$; thus the values taken on by context variables $y_1, \ldots, y_e$ in $S_i$ and in $S_j$ have no effect on the executability of path $p$ starting from $S_i$ and starting from $S_j$. Once this is established, we may conclude that path $p$ is also executable starting from $S_j$ since $S_i$ and $S_j$ are different from each other only in the values taken on by $u_1, \ldots, u_a$, and $y_1, \ldots, y_e$; and those values do not affect the executability of path $p$ starting from $S_i$ and $S_j$.

On contrary to the above statement we wish to prove, suppose context variables $y_{k_1}, \ldots, y_{k_m}$ in $\{y_1, \ldots, y_e\}$ are referenced before they are possibly assigned along path $p$, among which $y$ is referenced earliest. The values taken on by the context variables from $\{y_1, \ldots, y_e\} - \{y_{k_1}, \ldots, y_{k_m}\}$ in $S_i$ and $S_j$ do not affect the executability of path $p$ starting from $S_i$ and $S_j$ because these context variables are not referenced until they are assigned along path $p$. Moreover, no other context variables from $\{y_{k_1}, \ldots, y_{k_m}\}$ are referenced before $y$ is first referenced along path $p$; therefore, $p$ is executable up to the point when $y$ is first
referenced, starting from $S_j$. Let the initial subpath of $p$ up to the point when $y$ is first referenced be $p_y$. If $y$ is undefined in $S_j$, then it must be undefined at the point when it is first referenced since the first assignment to $y$ along $p$ would be after that point. This violates the basic assumption. If $y$ is defined in $S_j$, due to the fact $y$ takes on different values in $S_i$ and $S_j$ and by the definition of the set $SS$, $y$ must take on the undefined value in one of the global state matrices in $SS$, say $S_n$, where $n \neq i, j$. Path $p_y$ cannot be executable starting from $S_n$, for otherwise $y$ will be undefined when it is first referenced at the end of path $p_y$, thereby leading to a violation of the basic assumption. Thus, there must exist some context variables in $\{x_1, \ldots, x_b\}$ that take on different values in $S_i$ and in $S_n$, and they are referenced before being assigned along $p_y$. Let $x$ be such a context variable that is referenced earliest along $p_y$. Then $p_y$ is executable up to the point when $x$ is first referenced, starting from $S_n$. Let $p_yx$ be the initial subpath of $p_y$ up to the point when $x$ is first referenced. Due to the fact $x$ takes on different values in $S_i$ and $S_n$ and by the definition of the set $SS$, $x$ must take on the undefined value in one of the global state matrices in $SS$, say $S_q$, where $q \neq i, j, n$. The previous argument can then be repeated until we are sure that there must exist a context variable that is undefined in a global state matrix in $SS$ and along $p$ the context variable is referenced before being assigned, and that $p$ is executable up to the point when the context variable is first referenced. However, the existence of such a context variable contradicts
the basic assumption. The repetition of the previous argument can be terminated as shown below. Each time the previous argument is applied, we are forced to consider a global state matrix in $SS$ and context variables, such as $y$ and $x$, that have not yet been considered in all previous applications of the argument. Thus, eventually we will go through every global state matrix in $SS$ or every context variable and then terminate the argument.

There is a problem with implementing the second reduction technique. We may need to generate every global state in an equivalence class before we can determine whether these global states indeed form an equivalence class, as can be seen from the example in the paragraph prior to the proof of Theorem 2. As a result, to implement the second reduction technique, it may require the generation of the whole global state graph before the reduction can actually proceed. Such a reduction turns out to be meaningless due to the already consumed time and space of generating the whole global state graph. A meaningful global state graph reduction technique for protocol validation should be one that reduce the global state graph 'on the fly' (during the global state graph generation). Consequently, we use a less general global state equivalence depending on the order in which global states are generated, and implement the second reduction technique in our protocol validation system as follows. Whenever a global state, say $S$, is generated, it is compared to the global states that have already been generated so far. If $S$ is found to be different from an already-generated global
state, say $S'$, only in the values taken on by context variables, say $x_1, \ldots, x_n$, that are undefined in either $S$ or $S'$, we drop $S$ and make the values taken on by $x_1, \ldots, x_n$ in $S'$ to be undefined. Using this reduction technique alone for the ABP specification given in Fig. 6.2, our protocol validation system generate only 66 reachable global states, as compared to 158 global states that would be generated without using any reduction techniques.

Note that the first reduction technique as presented in Section 7.1 determines the equivalence of global states using only the local data flow information of each FSM independently without relying on the interaction between entities. On the other hand, the second reduction technique presented here determines the equivalence of global states using the presence of undefined values taken on by context variables, which in turn relies heavily on the interaction between entities. In some situation, the two reduction techniques may overlap in the sense that two global state matrices are considered equivalent by both reduction techniques. For the example in Fig. 6.2, all occurrences of global state equivalence considered by the second reduction technique happens to be considered as equivalent by the first reduction technique. But it by no means implies that one subsumes the other. Rather, the two complement to each other. Therefore, we adopt the union of the global state equivalences defined in both reduction techniques in our protocol validation system to achieve a higher reduction ratio.
In passing, we make a comparison between our global state graph reduction techniques and a technique called system state analysis recently proposed by Lundy and Miller [29]. They analyzed a general data transfer protocol specified in a model similar to the EFSM model using system state analysis. System state analysis actually generates a system state graph that is much smaller than the global state graph. This technique, however, is only applicable to the particular protocol used in their paper [29]. On the other hand, our global state graph reduction techniques are based on the model itself and do not depend on any features of protocols; Therefore, we claim that our techniques are much more general than theirs.

7.4 Experimental Results for a Subset of Stenning's Protocol

To experiment with our global state graph reduction techniques, we specify a subset of Stenning's data transfer protocol [44] in the ETG protocol specification language and then run our protocol validation system for it. Fig. 7.5 contain the results we have obtained, which indicate the real merits of our global state graph reduction techniques. In this figure, we demonstrate the results of running our protocol validation system with no reduction techniques included, with only the first reduction technique included,
Figure 7.5. Results of experimenting with our global state graph reduction techniques for a subset of Stenning’s protocol.
with only the second reduction technique included, and with both reduction techniques included. For different window sizes, Stenning's protocol allows a different maximal number of messages flowing through each channel. We fix the receiver window size to be one and adjust the sender window size to obtain a different maximal number of messages flowing through a channel. Then for each of these different settings, different reduction ratios are found, all of which give us more confidence in using our global state graph reduction techniques.
CHAPTER VIII

Conclusion and Future Research

This chapter summarizes the main results made by this research and suggests some future work because of the limitations of our approach to protocol synthesis and analysis.

8.1 Conclusion

This dissertation is concerned about two major aspects of Protocol Engineering [34]: protocol synthesis and protocol analysis. In the part of protocol synthesis, we propose the use of a state-transition model for specifying communication services and protocols. A protocol derivation algorithm is then given to produce a protocol specification from a given service specification. To complement the protocol derivation in dealing with error-prone communication media, an error-recovery transformation procedure is further devised to transform any protocol specification produced from the
protocol derivation algorithm into a corresponding error-recoverable one.

We regard the construction of protocol specifications from requirement specifications (which might be called the most abstract service specifications) as a procedure that searches through the refinement tree [23] to find out the most appropriate protocol specification, which is located at the bottom (or leaf) level of the refinement tree (see Fig. 8.1). The search procedure should be very knowledge-intensive in order to be efficient. The research in the literature regarding the cross-fertilization of Artificial Intelligence and Software Engineering [43] may shed some light on the procedure. The algorithm developed in the paper may be considered as the refinement of the most detailed service specifications, which sit one level above the leaf level, to obtain the protocol specifications. The error-recovery transformation can be treated as the product of interaction between the 'error-recovery' knowledge and the efficiency knowledge, taking into account the fixed patterns of the generated synchronizing protocol FSMs.

In the part of protocol analysis, reachability analysis is considered as a means for protocol validation/verification. Nevertheless, reachability analysis is often plagued by the global state state explosion problem. In order to alleviate this problem for protocol validation/verification using reachability analysis in the EFSoM model, two global state graph reduction techniques are proposed. These reduction techniques have shown a significant effect on reducing the global state graph for the ABP protocol. ABP is a
Figure 8.1. Refinement tree for protocol construction.
protocol in which the range of possible values that could be taken on by a context variable is not wide. For protocols with wide ranges of possible values that could be taken on by context variables, the effect of reduction can be more extensive, as can be expected from the upper bound of the reduction ratio given in Subsection 7.1.2.

One might relate our protocol derivation algorithm to the global state graph reduction techniques in the following way (see Fig. 8.2). A given service specification as a set of directly coupled FSMs may be the input to our protocol derivation algorithm for producing a protocol specification, which is two sets of directly coupled FSMs and serves as a _draft design_ for the desired protocol specification. The draft-design protocol specification may then be augmented with some context variables, predicates on context variables, and operations on context variables to meet the requirements of the desired protocol specification. Consequently, the resultant protocol specification becomes two sets of directly coupled FSMs whose state transitions are associated with access to context variables (called EFSMs for short). Although the draft-design protocol specification is guaranteed to be free from any logical errors, the desired protocol specification, which is different from the draft-design protocol specification by containing context variables, needs to be validated to reveal any logical errors. For the purpose of validation, each set of directly coupled EFSMs in the draft-design protocol specification can then be transformed into an EFSM using a reachability algorithm, taking direct coupling into consideration. After that, the global state graph
Figure 8.2. A possible use of our protocol derivation algorithm and global state graph reduction techniques.
reduction techniques may be used in the reachability analysis to validate the desired protocol specification. In case any logical errors are found, the protocol designer needs to modify the protocol specification in order to get rid of the errors. However, the effect of the modification needs to be checked by applying reachability analysis to see whether there are still some logical errors in the protocol specification. The procedure repeats until no logical errors can be found in the protocol specification. The rectangular boxes in Fig. 8.2 represents the portions that can be automatically done using the algorithms provided in this research, whereas the oval boxes are the portions that have to be done by human protocol designers.

8.2 Limitations and Future Research

In the part of protocol synthesis, several limitations on our approach can be identified and itemized as follows. Based on the limitations, future research to remove the limitations is also discussed.

- A high degree of concurrency in the execution of service primitives can be achieved in our model by running service FSMs in parallel. Even so, any synchronizing protocol FSM pair produced by the protocol derivation algorithm is always closely synchronized in the sense that the communication pattern of the
synchronization messages is 'handshaking', there are no message collisions, and at most two messages are in transit at any instant for the synchronizing protocol FSM pair. Therefore, the expressive power of our state-transition model is still limited as far as the control aspect of protocols is concerned. The study of an appropriate way to enhance the expressive power of our model is identified as a direction for future research.

- For modeling real-life protocols, the addition of parameter, variable and time specifications to our service model is mandatory. However, the addition may have an extensive impact on the protocol derivation algorithm, requiring more careful investigation.

- The optimization issue for communication protocols raised in [7], [30], and [36] is still an open question. The issue in our context for either error-free protocols or error-recoverable protocols is also a challenging work. Two points about the optimization of the generated error-recoverable protocol specifications are identified, i.e., the elimination of redundant timers and the use of negative acknowledgments. The discussion on them follows.

  - Let us consider the optimization issue on a transformed error-recoverable protocol specification. If we make stronger the fairness assumption about the communication media, some timers may become redundant and can thus be eliminated. For example, several versions of the Alternating Bit
Protocol (ABP) only use a timer in the sender for retransmission of lost messages. But our error-recovery transformation would require a timer for each of the sender and the receiver. However, the fairness assumptions about the communication media in these ABP protocol specifications and our ABP protocol specification are not the same. They assume that the communication media will correctly deliver a message infinitely often if the message is retransmitted an infinite number of times. On the other hand, our assumption is that the communication media will correctly deliver a message at least once if the message is retransmitted an infinite number of times. Obviously, their assumption is stronger than ours, thus making the use of a timer only in the sender justifiable. In case we also make our assumption as strong as theirs, we should be able to remove the timer in the receiver without sacrificing the functionality of the protocol. At this moment, we still don’t have a general solution for eliminating redundant timers from any produced error-recoverable protocol specification if the fairness assumption about the communication media is made stronger.

The use of negative acknowledgments in an error-recoverable protocol may reduce the time period between two consecutive transmissions of a same message, thereby increasing the average throughput of message delivery between two service users. However, it also complicates protocols and introduces some processing overhead. We believe that the use of negative
Acknowledgments should depend on the actual environments in which the protocol will be implemented. In case we do wish to use negative acknowledgments in our error-recoverable protocol specifications, it is interesting to study the right way to include them in the specification.

In the part of protocol analysis, the global state graph reduction techniques not only can be used for protocol validation, but are also applicable to protocol verification, when enriched with an appropriate bookkeeping routine such as the one given in Subsection 7.2.1 for recording the snapshot sets. These reduction techniques can alleviate the global state explosion problem, which is caused in part by a wide range of possible values that could be taken on by a context variable. However, more research is needed to further reduce the global state graph so that the global state explosion problem can be further alleviated. One of the aspects that are yet to be considered in the global state graph reduction techniques proposed in this dissertation is the relation between the values of context variables and the channel contents. The global state graph reduction techniques we propose have exploited either the data flow information within entity specifications or the assumption that no undefined variables are allowed to be referenced. However, none of them addresses the relation between the values of context variables and the channel contents. To take this aspect into consideration, the symbolic evaluation techniques [32] are expected to play an important role. The closed covers technique [13] is another candidate. We have observed a connection
between our global state graph reduction techniques and temporal logic as indicated in Appendix C. Hopefully, this connection may be utilized to facilitate further reduction in global state graph for protocol validation and verification.
Appendix A

Syntax of the ETG Protocol Specification Language

A portion of this appendix is derived from Appendix A of C. S. Lu's dissertation [28], which explains the syntax of the TG (Transmission Grammar) protocol specification language.

In the ETG protocol validation system, a protocol specification is written according to the syntax given in Figure A.1. The Backus-Naur Form (BNF) notation [33] is adopted for describing the syntax. Non-terminals, which represent syntactic classes in the syntax, are enclosed in angle brackets; terminals, which form the symbols that need to appear in the protocol specification, are set off by double quotes. Alternative production rules are separated by vertical bars. In the following paragraphs, which are numbered to correspond to the production rules of Figure A.1, we discuss this syntax in detail.
1. A <protocol-specification> is the equivalent of an <entity-specs>. Due to the assumption made in the ETG protocol validation system, i.e. communication media are implicitly modeled as error-free First-In-First-Out (FIFO) queues, the designer need not specify the communication channels in the communication system.

2. An <entity-specs> may consist of two or more <entity-specification>’s, one for each entity, since a single entity specification does not constitute a communication system.

3. An <entity-process> is composed of three fields in order, an <entity-header>, an <entity-proper> and a <terminator>.

4. An <entity-header> is used to declare sequence variables (or context variables) for the entity. It begins with the reserved word "$SEQ_VAR", followed by a list of sequence variable declarations, denoted as <var-dec-list>. It may be also legitimate to simply have a null <entity-header>, indicating that no context variables are needed in the entity specification.

5. A <var-dec-list> is a list of one or more <var-dec>, separated by commas.

6. A <var-dec> begins with an integer interval, followed by a variable name. This integer interval specifies all possible values taken on by the variable besides the
undefined value. It is also legitimate to have a variable carry an initial value, assigned through "=" in the <var-dec>.

7. An <entity-proper> is a list of one or more production rules. All the production rules corresponding to a non-terminal need be specified in shorthand and are referred to as <entity-productions-for-non-terminal>.

8. <entity-productions-for-non-terminal> specifies, in shorthand, all the alternative production rules corresponding to a non-terminal, which begin with the <non-terminal>, followed by "::=" and an <entity-production-expression>, and end with a period. Two adjacent syntactic items are separated by a <delimiter>.

9. An <entity-production-expression> is a sequence of one or more <entity-production-terms>. Any two consecutive production terms are separated by a comma.

10. An <entity-production-term> represents the label of a state transition and the next state following the transition in the EFSM model. The label of a state transition contains two parts, namely <condition> and <action>. The next state is denoted as a <non-terminal>.

11. A <condition> is a sequence of predicates and/or receive events. It may also be null, expressing a vacuously TRUE condition.
12. A `<predicate-or-receive>` is either a `<predicate>` or a `<receive>`.

13. Four predicate comparators are currently available in the ETG protocol specification language, viz. equality, inequality, less-than, and greater-than. The TRUE/FALSE value of the predicate is obtained by comparing the value taken on by the `<variable>` and that taken on by `<simple-expression>` according to the semantics of the comparator.

14. An `<action>` is a sequence of operations and/or send events. It may also be null, expressing a no-operation action.

15. Three operation operators are currently available in the ETG protocol specification language, viz. increment, decrement, and assignment. The effects of the operations with `+`, `−`, and `A` operators are incrementing the value taken on by `<variable>` by the value taken on by `<simple-expression>`, decrementing the value taken on by `<variable>` by the value taken on by `<simple-expression>`, and assigning to `<variable>` the value taken on by `<simple-expression>`, respectively.

16. A receive event is denoted as "D.", followed by the entity number of the source, a ".", and then `<message-dequeued>`.

17. A send event is denoted as "Q.", followed by the entity number of the desti-
nations, a ".", and then <message-queued>.

18. A <message-dequeued> is either just a <message-type> or a <message-type> followed by a <variable-list> delimited within a pair of parentheses.

19. A <message-queued> is either just a <message-type> or a <message-type> followed by an <expression-list> delimited within a pair of parentheses.

20. An <expression-list> is a list of one or more <simple-expression>, separated by commas.

21. A <variable-list> is a list of one or more <variable>, separated by commas.

22. A <simple-expression> is either just a <variable>, a <variable> plus an <integer>, or a <variable> minus an <integer>.

23. A <multi-entity-no> is a sequence of one or more <entity-no>.

24. An <entity-no> may be any numeric digit from 1 through the number of communicating entities in the validation; the number representing the entity to which the send or receive event belongs cannot be used as the <entity-number> of this operation because an entity should not, for instance, send a message to or receive a message from itself. Currently, the maximum number of entities is 9 because the
entity identification field in a validation operation is limited by one digit; the maximum number could be expanded up to 15 if the hexadecimal, instead of the decimal, number system were adopted. Digit 0 represents the multi-entities of a broadcast message, i.e. all of the entities (of the communication system in the validation) other than the entity from which this broadcast message originates.

25. A <message> is a sequence of one or more alphanumeric characters. Upper case letters are treated differently from their corresponding lower case ones. The maximum number of characters in a message is limited by parameter SALPHA in the program.

26. A <non-terminal> is a sequence of one or more alphanumeric characters. Upper case letters are treated differently from their corresponding lower case ones. The maximum number of characters in a non-terminal is limited by parameter LALPHA in the program. LALPHA is the maximum size of a terminal or non-terminal; therefore, the size of an identifier for a terminal or non-terminal is LALPHA or LALPHA-2, respectively. (Remember a non-terminal is enclosed in a pair of angle brackets.)

27. A <variable> is a sequence of one or more alphanumeric characters.

28. A <delimiter> is a sequence of one-or-more space or newline characters.
29. A `<terminator>` is a sequence of one or more dashes ("-"). It signifies the end of the specification for an entity process.
1. \(<\text{protocol-specification}>\)  
   ::= \(<\text{entity-specs}>\)

2. \(<\text{entity-specs}>\)  
   ::= \(<\text{entity-specification}>\) \(<\text{entity-specs}>\) 
     | \(<\text{entity-specification}>\)

3. \(<\text{entity-specification}>\)  
   ::= \(<\text{entity-header}>\) \(<\text{entity-proper}>\) \(<\text{terminator}>\)

4. \(<\text{entity-header}>\)  
   ::= \("\text{SEQ_VAR}\" \(<\text{delimiter}>\) \(<\text{var-dec-list}>\) \(<\text{delimiter}>\) \("\).\) 
     | \(\varepsilon\)

5. \(<\text{var-dec-list}>\)  
   ::= \(<\text{var-dec}>\) \(<\text{delimiter}>\) \(",\) \(<\text{var-dec-list}>\) 
     | \(<\text{var-dec}>\)

6. \(<\text{var-dec}>\)  
   ::= \("[\" <\text{integer}> \," <\text{integer}> \"]\" <\text{variable}> 
     | \("[\" <\text{integer}> \," <\text{integer}> \"]\" <\text{variable}> \:" <\text{integer}>\)

7. \(<\text{entity-proper}>\)  
   ::= \(<\text{entity-productions-for-non-terminal}>\) \(<\text{entity-proper}>\) 
     | \(<\text{entity-productions-for-non-terminal}>\)

8. \(<\text{entity-productions-for-non-terminal}>\)  
   ::= \(<\text{non-terminal}>\) \(<\text{delimiter}>\) \("::=\) \(<\text{delimiter}>\) 
     \(<\text{entity-production-expression}>\) \(<\text{delimiter}>\) \(";\)"

---

**Figure A.1.** Syntax of the ETG protocol specification language.
9. <entity-production-expression>
   ::= <entity-production-term> <delimiter> ",” <delimiter>
       <next-entity-production-expression>
       | <entity-production-term>

10. <entity-production-term>
    ::= <condition> <delimiter> <action>
        <delimiter> <non-terminal>

11. <condition>
    ::= <predicate-or-receive> <delimiter> <condition>
        | ε

12. <predicate-or-receive>
    ::= <predicate>
        | <receive>

13. <predicate>
    ::= "=.” <variable> ".” <simple-expression>
        "#.” <variable> ".” <simple-expression>
        "<.” <variable> ".” <simple-expression>
        ">.” <variable> ".” <simple-expression>

14. <action>
    ::= <operation-or-send> <delimiter> <action>
        | ε

15. <operation>
    ::= "+.” <variable> ".” <simple-expression>
        "-.” <variable> ".” <simple-expression>
        "A.” <variable> ".” <simple-expression>

Figure A.1. continued.
16. \(<\text{receive}\>\)
\[::= \text{"D." } <\text{entity-no}> \text{"." } <\text{message-dequeued}>\]

17. \(<\text{send}\>\)
\[::= \text{"Q." } <\text{multi-entity-no}> \text{"." } <\text{message-queued}>\]

18. \(<\text{message-dequeued}\>\)
\[::= <\text{message-type}> \]
\[\text{ | } <\text{message-type}> \text{"(" } <\text{variable-list}> \text{")"}\]

19. \(<\text{message-queued}\>\)
\[::= <\text{message-type}> \]
\[\text{ | } <\text{message-type}> \text{"(" } <\text{expression-list}> \text{")"}\]

20. \(<\text{expression-list}\>\)
\[::= <\text{simple-expression}> \text{"," } <\text{expression-list}> \]
\[\text{ | } <\text{simple-expression}>\]

21. \(<\text{variable-list}\>\)
\[::= <\text{variable}> \text{"," } <\text{variable-list}> \]
\[\text{ | } <\text{variable}>\]

22. \(<\text{simple-expression}\>\)
\[::= <\text{variable}> \]
\[\text{ | } <\text{variable}> \text{"+" } <\text{integer}> \]
\[\text{ | } <\text{variable}> \text{"-" } <\text{integer}>\]

23. \(<\text{multi-entity-no}\>\)
\[::= <\text{entity-no}> <\text{multi-entity-no}> \]
\[\text{ | } <\text{entity-no}>\]

---

Figure A.1. continued.
24. <entity-no>
   ::= "0"|"1"|"2"|"3"|"4"
   ::= "5"|"6"|"7"|"8"|"9"

25. <message-type>
    ::= <sequence-of-characters>

26. <non-terminal>
    ::= <sequence-of-characters>

27. <variable>
    ::= <sequence-of-characters>

28. <delimiter>
    ::= <sequence-of-spaces-or-newlinechars>

29. <terminator>
    ::= <sequence-of-dashes>

Figure A.1. continued and concluded.
Appendix B

Diagnosis Report on Running the ETG Protocol Validation System for the Alternating Bit Protocol

The following is an abridged diagnosis report on running the ETG protocol validation system for the Alternating Bit Protocol (ABP) shown in Fig. 6.2. The notation ‘---’ represents an empty channel and ‘xxx’ a channel which is not available for use by any ‘Q’ or ‘D’ action in a protocol specification. A question mark ‘?’ is used to denote an undefined value. A list delimited within a pair of ‘{’ and ‘}’ represents the values taken on by context variables declared for the corresponding entity, and a vertical bar ‘|’ separates every two adjacent values. The first value in the list records the value for the context variable declared first and the second value for that declared the second, and so on.
The ETG Protocol Validation System

by

P. M. Chu

$\text{SEQ â€œ} [0..1] X = 0$, $[0..1] Y$.

$\langle 1 \rangle ::= \text{new} \langle 2 \rangle$.

$\langle 2 \rangle ::= Q.2.D(X) \langle 3 \rangle$.

$\langle 3 \rangle ::= D.4.A(Y) = Y.X + X.1 \langle 1 \rangle$,

$D.4.A(Y) # Y.X \langle 2 \rangle$,

$D.4.Err \langle 2 \rangle$.

$\text{SEQ â€œ} [0..1] X$.

$\langle 1 \rangle ::= D.1.D(X) \langle 2 \rangle$.

$\langle 2 \rangle ::= Q.3.D(X) \langle 1 \rangle$,

$Q.3.D(X+1) \langle 1 \rangle$.
\texttt{SEQ_VAR} [0..1] X = 0 ,
\texttt{SEQ_VAR} [0..1] Y .

\texttt{<1> := D.2.D(Y) = .Y.X + .X.1 <2> ,}
\texttt{D.2.D(Y) # .Y.X <3> ,}
\texttt{D.2.Err <3> .}

\texttt{<2> := use <3> .}
\texttt{<3> := Q.4.A(X-1) <1> .}

\texttt{SEQ_VAR [0..1] X .}

\texttt{<1> := D.3.A(X) <2> .}
\texttt{<2> := Q.1.A(X) <1> ,}
\texttt{Q.1.A(X+1) <1> ,}
\texttt{Q.1.Err <1> .}
**** Data flow analysis completed.

<table>
<thead>
<tr>
<th>Entity</th>
<th>State</th>
<th>Dead variable set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;1&gt; Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;2&gt; Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3&gt; Y</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>&lt;1&gt; X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;2&gt;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>&lt;1&gt; Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;2&gt; Y</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;3&gt; Y</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;1&gt; X</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;2&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**** Global state graph generation started.

---- Expand state 1

\[
1 <1> ::= \text{new} <2> \leftarrow \text{Ent 1}
\]

\[
<2> ::= \text{Q.2.D(X)} <3> \leftarrow \text{Ent 1}
\]
Add state  3

**** Expand state  3 ( previous state )

3  \( <1> ::= D.1.D(X) <2> \) <- Ent 2

\[
\begin{array}{ccc}
\text{xxx} & \text{\{} & \text{?} \\
\text{xxx} & \text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{---} & \text{xxx} & \text{xxx} & \text{\{} & \text{?}
\end{array}
\]

Add state  4

**** Expand state  4 ( previous state )

4  \( <2> ::= Q.3.D(X) <1> \) <- Ent 2

\[
\begin{array}{ccc}
\text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{---} & \text{xxx} & \text{xxx} & \text{\{} & \text{?}
\end{array}
\]

Add state  5

5  \( <2> ::= Q.3.D(X+1) <1> \) <- Ent 2

\[
\begin{array}{ccc}
\text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{---} & \text{xxx} & \text{xxx} & \text{\{} & \text{?}
\end{array}
\]

Add state  6

6  \( <2> ::= Q.3.Err <1> \) <- Ent 2

\[
\begin{array}{ccc}
\text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{xxx} & \text{\{} & \text{0} & \text{?} \\
\text{---} & \text{xxx} & \text{xxx} & \text{\{} & \text{?}
\end{array}
\]

Add state  7
---- Expand state 7 (previous state)

7 \( <1> ::= D.2.Err <3> \leftarrow \text{Ent 3} \)

\[
\begin{array}{ccc}
<3>\{0|?\} & \text{---} & \text{xxx} \\
\text{xxx} & <1>\{0\} & \text{---} \\
\text{xxx} & \text{xxx} & <3>\{0|?\} \\
\text{---} & \text{xxx} & <1>\{?\}
\end{array}
\]

Add state 8

---- Expand state 8 (previous state)

8 \( <3> ::= Q.4.A(X-1) <1> \leftarrow \text{Ent 3} \)

\[
\begin{array}{ccc}
<3>\{0|?\} & \text{---} & \text{xxx} \\
\text{xxx} & <1>\{0\} & \text{---} \\
\text{xxx} & \text{xxx} & <1>\{0|?\} \\
\text{---} & \text{xxx} & <1>\{?\}
\end{array}
\]

Add state 9

---- Expand state 9 (previous state)

9 \( <1> ::= D.3.A(X) <2> \leftarrow \text{Ent 4} \)

\[
\begin{array}{ccc}
<3>\{0|?\} & \text{---} & \text{xxx} \\
\text{xxx} & <1>\{0\} & \text{---} \\
\text{xxx} & \text{xxx} & <1>\{0|?\} \\
\text{---} & \text{xxx} & <2>\{1\}
\end{array}
\]

Add state 10

---- Expand state 10 (previous state)

10 \( <2> ::= Q.1.A(X) <1> \leftarrow \text{Ent 4} \)

\[
\begin{array}{ccc}
<3>\{0|?\} & \text{---} & \text{xxx} \\
\text{xxx} & <1>\{0\} & \text{---} \\
\text{xxx} & \text{xxx} & <1>\{0|?\} \\
\text{A(1)} & \text{xxx} & <1>\{1\}
\end{array}
\]

Add state 11

11 \( <2> ::= Q.1.A(X+1) <1> \leftarrow \text{Ent 4} \)
\[<3>|0|?\]  \[<1>|0\]  \[<1>|0|?\]  \[A(0)\]  \[Err\]  

Add state  12

\[
\begin{array}{c|c|c}
12 & ::= & Q.1.Err <1> <- Ent 4 \\
13 & ::= & D.4.Err <2> <- Ent 1 \\
14 & ::= & D.4.A(Y) = .Y.X + .X.1 <1> <- Ent 1 \\
\end{array}
\]

Same as state  2

Add state  14

---- Expand state  14 (previous state)
15  \( \langle 1 \rangle ::= \text{new} \langle 2 \rangle \quad \langle \text{Ent 1} \rangle \)

\[
\begin{array}{ccc}
\langle 2 \rangle \{1|0\} & \text{---} & \text{xxx} \\
\text{xxx} & \langle 1 \rangle \{0\} & \text{---} \\
\text{xxx} & \langle 1 \rangle \{0|?\} & \text{---} \\
\text{---} & \text{xxx} & \langle 1 \rangle \{1\}
\end{array}
\]

Add state 15

---- Expand state 15 (previous state)

16  \( \langle 2 \rangle ::= \text{Q.2.D(X)} \langle 3 \rangle \quad \langle \text{Ent 1} \rangle \)

\[
\begin{array}{ccc}
\langle 3 \rangle \{1|0\} & \text{D(1)} & \text{xxx} \\
\text{xxx} & \langle 1 \rangle \{0\} & \text{---} \\
\text{xxx} & \langle 1 \rangle \{0|?\} & \text{---} \\
\text{---} & \text{xxx} & \langle 1 \rangle \{1\}
\end{array}
\]

Add state 16

---- Expand state 16 (previous state)

17  \( \langle 1 \rangle ::= \text{D.1.D(X)} \langle 2 \rangle \quad \langle \text{Ent 2} \rangle \)

\[
\begin{array}{ccc}
\langle 3 \rangle \{1|0\} & \text{---} & \text{xxx} \\
\text{xxx} & \langle 2 \rangle \{1\} & \text{---} \\
\text{xxx} & \langle 1 \rangle \{0|?\} & \text{---} \\
\text{---} & \text{xxx} & \langle 1 \rangle \{1\}
\end{array}
\]

Add state 17

---- Expand state 17 (previous state)

18  \( \langle 2 \rangle ::= \text{Q.3.D(X)} \langle 1 \rangle \quad \langle \text{Ent 2} \rangle \)

\[
\begin{array}{ccc}
\langle 3 \rangle \{1|0\} & \text{---} & \text{xxx} \\
\text{xxx} & \langle 1 \rangle \{1\} & \text{D(1)} \\
\text{xxx} & \langle 1 \rangle \{0|?\} & \text{---} \\
\text{---} & \text{xxx} & \langle 1 \rangle \{1\}
\end{array}
\]

Add state 18

19  \( \langle 2 \rangle ::= \text{Q.3.D(X+1)} \langle 1 \rangle \quad \langle \text{Ent 2} \rangle \)
<3>{1|0}  ---  xxx  xxx
xxx  <1>{1}  D(0)  xxx
xxx  xxx  <1>{0|?}  ---
---  xxx  xxx  <1>{1}
Add state 19

20  <2> ::= Q.3.Err <1>  <- Ent 2

<3>{1|0}  ---  xxx  xxx
xxx  <1>{1}  Err  xxx
xxx  xxx  <1>{0|?}  ---
---  xxx  xxx  <1>{1}
Add state 20

---- Expand state 20 (previous state)

21  <1> ::= D.2.Err <3>  <- Ent 3

<3>{1|0}  ---  xxx  xxx
xxx  <1>{1}  ---  xxx
xxx  xxx  <3>{0|?}  ---
---  xxx  xxx  <1>{1}
Add state 21

---- Expand state 21 (previous state)

22  <3> ::= Q.4.A(X-1) <1>  <- Ent 3

<3>{1|0}  ---  xxx  xxx
xxx  <1>{1}  ---  xxx
xxx  xxx  <1>{0|?}  A(1)
---  xxx  xxx  <1>{1}
Add state 22

---- Expand state 22 (previous state)

23  <1> ::= D.3.A(X) <2>  <- Ent 4

<3>{1|0}  ---  xxx  xxx
xxx  <1>{1}  ---  xxx
xxx  xxx  <1>{0|?}  ---
Add state 23

---- Expand state 23 ( previous state )

24 \( <2> ::= \text{Q.1.A}(X) <1> \) <- Ent 4
\[
\begin{array}{c|c|c|c|c}
<3>{1|0} & \text{---} & \text{xxx} & \text{xxx} \\
\hline
\text{xxx} & <1>{1} & \text{---} & \text{xxx} \\
\text{xxx} & \text{xxx} & <1>{0|?} & \text{---} \\
A(1) & \text{xxx} & \text{xxx} & <1>{1} \\
\end{array}
\]
Add state 24

25 \( <2> ::= \text{Q.1.A}(X+1) <1> \) <- Ent 4
\[
\begin{array}{c|c|c|c|c}
<3>{1|0} & \text{---} & \text{xxx} & \text{xxx} \\
\hline
\text{xxx} & <1>{1} & \text{---} & \text{xxx} \\
\text{xxx} & \text{xxx} & <1>{0|?} & \text{---} \\
A(0) & \text{xxx} & \text{xxx} & <1>{1} \\
\end{array}
\]
Add state 25

26 \( <2> ::= \text{Q.1.Err} <1> \) <- Ent 4
\[
\begin{array}{c|c|c|c|c}
<3>{1|0} & \text{---} & \text{xxx} & \text{xxx} \\
\hline
\text{xxx} & <1>{1} & \text{---} & \text{xxx} \\
\text{xxx} & \text{xxx} & <1>{0|?} & \text{---} \\
\text{Err} & \text{xxx} & \text{xxx} & <1>{1} \\
\end{array}
\]
Add state 26

---- Expand state 26 ( previous state )

27 \( <3> ::= \text{D.4.Err} <2> \) <- Ent 1
\[
\begin{array}{c|c|c|c|c}
<2>{1|0} & \text{---} & \text{xxx} & \text{xxx} \\
\hline
\text{xxx} & <1>{1} & \text{---} & \text{xxx} \\
\text{xxx} & \text{xxx} & <1>{0|?} & \text{---} \\
\text{---} & \text{xxx} & \text{xxx} & <1>{1} \\
\end{array}
\]
Same as state 15

---- Expand state 25
| <3>{1|0} | --- | xxx | xxx |
|---|---|---|---|
| xxx | <1>{1} | --- | xxx |
| xxx | xxx | <1>{0|?} | --- |
| A(0) | xxx | xxx | <1>{1} |

28 \( <3> ::= D.4.A(Y) \#.Y.X <2> \) <- Ent 1

| <3>{1|0} | --- | xxx | xxx |
|---|---|---|---|
| xxx | <1>{1} | --- | xxx |
| xxx | xxx | <1>{0|?} | --- |
| --- | xxx | xxx | <1>{1} |

Same as state 15

--- Expand state 24

| <3>{1|0} | --- | xxx | xxx |
|---|---|---|---|
| xxx | <1>{1} | --- | xxx |
| xxx | xxx | <1>{0|?} | --- |
| A(1) | xxx | xxx | <1>{1} |

29 \( <3> ::= D.4.A(Y) =.Y.X +.X.1 <1> \) <- Ent 1

| <1>{0|1} | --- | xxx | xxx |
|---|---|---|---|
| xxx | <1>{1} | --- | xxx |
| xxx | xxx | <1>{0|?} | --- |
| --- | xxx | xxx | <1>{1} |

Same as state 1

--- Expand state 19

| <3>{1|0} | --- | xxx | xxx |
|---|---|---|---|
| xxx | <1>{1} | D(0) | xxx |
| xxx | xxx | <1>{0|?} | --- |
| --- | xxx | xxx | <1>{1} |

30 \( <1> ::= D.2.D(Y) =.Y.X +.X.1 <2> \) <- Ent 3

| <3>{1|0} | --- | xxx | xxx |
|---|---|---|---|
| xxx | <1>{1} | --- | xxx |
| xxx | xxx | <2>{1|0} | --- |
| --- | xxx | xxx | <1>{1} |
Add state 27

---- Expand state 27 (previous state)

31 \( <2> ::= \text{use} <3> \leftarrow \text{Ent 3} \)

\[
\begin{array}{ccc}
\langle 3\rangle\{1|0\} & \text{---} & xxx & xxx \\
xxx & \langle 1\rangle\{1\} & \text{---} & xxx \\
xxx & xxx & \langle 3\rangle\{1|0\} & \text{---} \\
\text{---} & xxx & xxx & \langle 1\rangle\{1\}
\end{array}
\]

Add state 28

---- Expand state 28 (previous state)

32 \( <3> ::= \text{Q.4.A}(X-1) \langle 1\rangle \leftarrow \text{Ent 3} \)

\[
\begin{array}{ccc}
\langle 3\rangle\{1|0\} & \text{---} & xxx & xxx \\
xxx & \langle 1\rangle\{1\} & \text{---} & xxx \\
xxx & xxx & \langle 1\rangle\{1|0\} & A(0) \\
\text{---} & xxx & xxx & \langle 1\rangle\{1\}
\end{array}
\]

Add state 29

---- Expand state 29 (previous state)

33 \( <1> ::= \text{D.3.A}(X) \langle 2\rangle \leftarrow \text{Ent 4} \)

\[
\begin{array}{ccc}
\langle 3\rangle\{1|0\} & \text{---} & xxx & xxx \\
xxx & \langle 1\rangle\{1\} & \text{---} & xxx \\
xxx & xxx & \langle 1\rangle\{1|0\} & \text{---} \\
\text{---} & xxx & xxx & \langle 2\rangle\{0\}
\end{array}
\]

Add state 30

:\
:\
:

Global State List

---------------------------------------------

state 1

\( \langle 1\rangle\{0|?\} \text{---} xxx \text{ xxx} \)
state 2

\[<2\{0\}?> \quad \text{---} \quad \text{xxx} \quad \text{xxx}\]
\[\text{xxx} \quad <1\{0\}?> \quad \text{---} \quad \text{xxx}\]
\[\text{---} \quad \text{xxx} \quad \text{xxx} \quad <1\{0\}?> \quad \text{---}\]

state 3

\[<3\{0\}?> \quad D(0) \quad \text{xxx} \quad \text{xxx}\]
\[\text{xxx} \quad <1\{0\}?> \quad \text{---} \quad \text{xxx}\]
\[\text{---} \quad \text{xxx} \quad <1\{0\}?> \quad \text{---}\]

state 4

\[<3\{0\}?> \quad \text{---} \quad \text{xxx} \quad \text{xxx}\]
\[\text{xxx} \quad <2\{0\}?> \quad \text{---} \quad \text{xxx}\]
\[\text{---} \quad \text{xxx} \quad <1\{0\}?> \quad \text{---}\]

state 5

\[<3\{0\}?> \quad \text{---} \quad \text{xxx} \quad \text{xxx}\]
\[\text{xxx} \quad <1\{0\}?> \quad D(0) \quad \text{xxx}\]
\[\text{---} \quad \text{xxx} \quad <1\{0\}?> \quad \text{---}\]

state 6

\[<3\{0\}?> \quad \text{---} \quad \text{xxx} \quad \text{xxx}\]
\[\text{xxx} \quad <1\{0\}?> \quad D(1) \quad \text{xxx}\]
\[\text{---} \quad \text{xxx} \quad <1\{0\}?> \quad \text{---}\]

state 7
| State | <3>{0|?} | --- | xxx | xxx |
|-------|------------|------|------|------|
| 8     | xxx        | <1>{0} | Err  | xxx  |
|       | xxx        | xxx   | <1>{0|?} | --- |
| 9     | ---        | xxx   | xxx  | xxx  |
|       | xxx        | <1>{0} | ---  | xxx  |
|       | xxx        | xxx   | <3>{0|?} | --- |
|       | ---        | xxx   | xxx  | <1>{?} |
| 10    | <3>{0|?}   | ---   | xxx  | xxx  |
|       | xxx        | <1>{0} | ---  | xxx  |
|       | xxx        | <1>{0|?} | A(1) |
|       | ---        | xxx   | xxx  | <1>{?} |
| 11    | <3>{0|?}   | ---   | xxx  | xxx  |
|       | xxx        | <1>{0} | ---  | xxx  |
|       | xxx        | <1>{0|?} | --- |
|       | A(1)       | xxx   | xxx  | <2>{1} |
| 12    | <3>{0|?}   | ---   | xxx  | xxx  |
|       | xxx        | <1>{0} | ---  | xxx  |
|       | xxx        | <1>{0|?} | --- |
|       | A(0)       | xxx   | xxx  | <1>{1} |
| 13    | <3>{0|?}   | ---   | xxx  | xxx  |
|       | xxx        | <1>{0} | ---  | xxx  |
|       | xxx        | <1>{0|?} | --- |
|       | A(0)       | xxx   | xxx  | <1>{1} |
| State 19 | <3>{1|0} | --- | xxx | xxx |
|----------|-----------|------|-----|-----|
|          | xxx       | <1>{1} | D(0) | xxx |
|          | xxx       | xxx   | <1>{0|?} | --- |
|          | ---       | xxx   | xxx | <1>{1} |

| State 20 | <3>{1|0} | --- | xxx | xxx |
|----------|-----------|------|-----|-----|
|          | xxx       | <1>{1} | Err | xxx |
|          | xxx       | xxx   | <1>{0|?} | --- |
|          | ---       | xxx   | xxx | <1>{1} |

| State 21 | <3>{1|0} | --- | xxx | xxx |
|----------|-----------|------|-----|-----|
|          | xxx       | <1>{1} | --- | xxx |
|          | xxx       | xxx   | <3>{0|?} | --- |
|          | ---       | xxx   | xxx | <1>{1} |

| State 22 | <3>{1|0} | --- | xxx | xxx |
|----------|-----------|------|-----|-----|
|          | xxx       | <1>{1} | --- | xxx |
|          | xxx       | xxx   | <1>{0|?} | A(1) |
|          | ---       | xxx   | xxx | <1>{1} |

| State 23 | <3>{1|0} | --- | xxx | xxx |
|----------|-----------|------|-----|-----|
|          | xxx       | <1>{1} | --- | xxx |
|          | xxx       | xxx   | <1>{0|?} | --- |
|          | ---       | xxx   | xxx | <2>{1} |

| State 24 | <3>{1|0} | --- | xxx | xxx |
|----------|-----------|------|-----|-----|
|          | xxx       | <1>{1} | --- | xxx |
|          | xxx       | xxx   | <1>{0|?} | --- |
|          | A(1)      | xxx   | xxx | <1>{1} |
Global State Graph

---  xxx  xxx  <2>{0}---

1  2.
2  3.
3  4.
4  5,  6,  7.
5  6.
6  8.
7  8.
8  9.
9  10.
10  11,  12,  13.
11  2.
12  14.
13  2.
14  15.
15  16.
16  17.
17  18,  19,  20.
18  21.
19  27.
20  21.
21  22.
22  23.
23  24,  25,  26.
24  1.
25  15.
26  15.
27  28.
28  29.
29  30.
30  31,  32,  33.
31  34.
32  41.
33 34.
34 35.
35 36.
36 37, 38, 39.
37 40.
38 28.
39 28.
40 21.
41 42.
42 43.
43 44.
44 45, 46, 47.
45 48.
46 55.
47 48.
48 49.
49 50.
50 51, 52, 53.
51 54.
52 42.
53 42.
54 34.
55 8.
56 48.

Total Steps: 72.

Total States: 56.

No Redundant Rules.

No Reception Errors.

No Deadlock Errors.

No ChOv Errors.
Appendix C

Application of the CTL Model Checking Algorithm to Live Variable Analysis

The CTL (Computation Tree Logic) is propositional, branching-time temporal logic developed by Clarke et al. [10] to express assertions about temporal properties of systems. There is a model checking algorithm proposed by the same people to verify properties expressible in CTL for finite-state systems [10]. In this appendix, we show how we can apply the CTL model checking algorithm to solve the problem associated with live variable analysis.

Since our main concern is live variable analysis, we explain here only one combination of the two operators $E$ (Exist) and $U$ (Until) in CTL that is used for live variable analysis. The meaning of the formula $\langle M, s_0 \rangle \models f$ is that $f$ holds at state $s_0$ in structure $M$, where $f$ is a CTL formula. When the structure $M$ is understood, we simply
ing from state $s_0$ in structure $M$ in which $f$ holds until $g$ holds. Formally stated, there exists some path $(s_0, s_1, \ldots)$ such that $\exists i [i \geq 0 \land s_i \models g \land \forall j [0 \leq j < i \Rightarrow s_j \models f]]$.

The problem associated with live variable analysis is to decide whether a context variable $x$ is live at a state $v$. Let $\text{Live}(x)$ stand for that context variable $x$ is live at the concerned state. Let $\text{Ref}(x)$ and $\text{Asg}(x)$ denote the fact that context variable $x$ is being referenced and being assigned at the concerned state, respectively. Then $\text{Live}(x)$ can be defined in terms of $\text{Ref}(x)$ and $\text{Asg}(x)$ as the following equation shows.

$$v \models \text{Live}(x) \equiv v \models E[\neg \text{Asg}(x) U \text{Ref}(x)],$$

which reads "Context variable $x$ is live at state $v$ if and only if there exists a path starting from $v$ in which $x$ will not be assigned until it is referenced".

The above equation works well for a finite-state system in whose finite-state diagram each context variable $x$ can only be either referenced or assigned, but not be referenced and assigned simultaneously at any state. To deal with finite-state systems in whose finite-state diagram each state is associated with a sequence of references and assignments to context variables (called finite-state reference-assignment systems hereafter), the equation should be modified as follows. Let $\text{Live}(x)$ stand for that context variable $x$ is live at the beginning of the concerned state. Let $\text{FRef}(x)$ denote the fact that at the concerned state the first access to context variable $x$ is a reference.
Let \( Asg(x) \) denote the fact that at the concerned state context variable \( x \) is assigned. Then \( Live(x) \) can be defined in terms of \( Ref(x) \) and \( Asg(x) \) as the following equation shows.

\[
v \models Live(x) \equiv v \models E[\neg Asg(x) U FRef(x)],
\]

which reads "Context variable \( x \) is live at the beginning of state \( v \) if and only if there exists a path \((v_0, v_1, \ldots, v_k)\), where \( k \geq 0 \) and \( v_0 = v \), such that the first access to \( x \) at state \( v_k \) is referenced and \( x \) is not assigned at states \( v_0, v_1, \ldots, v_{k-1} \)."

In the EFSM model for protocol specification (more details can be found in Chapter VI of this dissertation), each protocol entity is specified as a Finite State Machine (FSM) along whose state transitions context variables can be accessed. However, context variables are not allowed to be accessed at states in the EFSM model. Before we can apply the CTL model checking algorithm to live variable analysis for an FSM in the EFSM model, we must first transform the FSM in the EFSM model to a finite-state reference-assignment system described in the previous paragraph. For each transition \( t \) from state \( u \) to state \( v \) in an FSM in the EFSM model, we simply create a state \( w \) labeled with appropriate \( FREF(x) \)'s and \( Asg(x) \)'s according to the label associated with \( t \). Then we create two transitions from \( u \) to \( w \) and from \( w \) to \( v \), respectively, to replace transition \( t \). State \( u \) and state \( w \) are left unlabeled. Shown in Fig. C.1 is an example of this transformation.
In the CTL model checking algorithm, formulas of the form $E[f U g]$ are handled as follows. First, we find all of those states that are labeled with $g$. We then trace back using the converse of the state transition relation to find all the states that can be reached by a path in which each state is labeled $f$. All such states should be labeled $E[f U g]$. It is possible that a state will be labeled $E[f U g]$ more than one time using the above algorithm. To avoid double work on the labeling, we need to quit the labeling procedure starting from one of the states labeled $g$ once a state labeled $E[f U g]$ is visited.

The following is the instantiation of the above algorithm for live variable analysis.

1. Let $S$ be the set of states labeled $\text{FRef}(x)$. Find $S$. 

Figure C.1. Transforming a transition in an FSM in the EFSM model to two consecutive transitions in a finite-state reference-assignment system.
2. For each state \( s \) in \( S \), do the following:

(a) Initialize a set \( OPEN \) to contain a single item, state \( s \).

(b) Do the following until \( OPEN \) becomes empty:

   i. Select and remove a state, say \( v \), from \( OPEN \).

   ii. For each immediate predecessor of state \( v \), say state \( u \), do the following:

       If state \( u \) is not labeled \( Live(x) \) nor labeled \( Asg(x) \), then label state \( u \) with \( Live(x) \) and add state \( u \) into \( OPEN \).

This algorithm will go through each state and transition at most once. Therefore, the time complexity of the algorithm would be \( O(n + d) \), where \( n \) and \( d \) are the numbers of states and transitions in the finite-state reference-assignment system, respectively.
Bibliography

Compiler Construction: Theory and Practice, Chapter 3. 

Note on Reliable Full-Duplex Transmission Over Half-Duplex Links. 

Finite State Description of Communication Protocols. 


Delay-Independent Design for Distributed Systems. 


Deriving Protocol Specifications from Service Specifications. 

Synthesis of Protocols for an Unlimited Number of Processes. 
A Sequence Method for Protocol Construction.  
In Proc. Sixth Int'l Workshop on Protocol Specification, Testing, and Verification,  

Automatic Verification of Finite-State Concurrent Systems using Temporal Logic  
Specifications.  
ACM Transactions on Programming Languages and Systems, 8(2):244–263,  
April 1986.

Petri Nets are Good for Protocols.  
74, June 1984.

Specification and Analysis of Concurrent Systems using STOCS Model.  

Closed Covers: To Verify Progress for Communicating Finite State Machines.  

Protocol Validation by Fair Progress State Exploration.  

Protocol Validation by Maximal Progress State Exploration.  

Synthesis of Communicating Finite-State Machines with Guaranteed Progress.  

Communicating Sequential Processes.  

Tracing Protocols.  

Protocol Verification Algorithm Using Reduced Reachability Analysis.

Component-Based Synthesis of Protocols for Unlimited Number of Processes.

[22] Y. Kakuda, Y. Wakahara, and M. Norigoe.
A New Algorithm for Fast Protocol Validation.

[23] E. Kant and D. R. Barstow.
Refinement Paradigm: The interaction of Coding and Efficiency Knowledge in Program Synthesis.

PDS - A Protocol Design System.


The Features and Facilities of ESTELLE.

CSP-based Specifications for Network Protocols and Services.

*Automated Validation of Communication Protocols*.
Specification and Analysis of a General Data Transfer Protocol.

On the Construction of Submodule Specifications and Communication Protocols.

Prentice-Hall, 1981.

Prentice-Hall, 1981.

Prentice-Hall, 1981.

[34] T. F. Piatkowski.
The State of the Art in Protocol Engineering.

An Algorithm to Construct Distributed Systems from State Machines.

Communication Protocol Synthesis.

An Implementation of an Automated Protocol Synthesizer (APS) and its Application to the X.21 Protocol.

[38] C. V. Ramamoorthy, Y. Yaw, W. T. Tsai, R. Aggarwal, and J. Song.
An Improved Protocol Validation Technique.

[40] B. G. Ryder and M. D. Carrol.
An Incremental Algorithm for Software Analysis.

[41] B. G. Ryder and M. C. Paull.
Incremental Data-Flow Analysis Algorithms.

Protocol Design Rules.

Research on Knowledge-Based Software Environments at Kestrel Institute.

A Data Transfer Protocol.

*Computer Networks*, Chapter 4.


The Importance of the Service Concept in the Design of Data Communications Protocols.

Reachability Analysis of Protocols with FIFO Channels.
Reachability Analysis of Protocols with Non-FIFO Channels.

Generalized Technique for Communication Protocol Validation.

Protocol Validation by Random State Exploration.

Towards Analyzing and Synthesizing Protocols.

An Interactive Protocol Synthesis Algorithm Using a Global State Transition Graph.

A Knowledge-Based System for Protocol Synthesis (KSPS).


[56] H. Zimmermann.