National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Language-Based Analysis of Communicating Finite State Machines

by

Jan Huus

M. C. S. Thesis

submitted to the School of Graduate Studies and Research

of the University of Ottawa

in partial fulfillment of the requirements for the degree of

Master of Computer Science*

Department of Computer Science

University of Ottawa

Ottawa, Ontario

*The Master of Computer Science program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science.

© Jan Huus, Ottawa, Canada, 1992

National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-80011-9

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ABSTRACT

Protocol validation has traditionally focused on detecting errors defined in terms of global states. A complementary approach is to analyze the process languages of a protocol, where a process language is defined as the set of executable event sequences of a process in the protocol. A general property of a process is "effectiveness", defined as the absence of unexecutable specified event sequences. A method is given for determining whether a specified process (the "host process") in a protocol is effective. The method involves generating a process language's finite state machine representation, called a process event graph (PEG). Every PEG node maps to a global state. Using the given method to generate a PEG, it is shown that, when the PEG generation does not detect error states, every reachable global state is reachable from a PEG node by an event sequence that does not include any host events. This introduces the possibility of a parallel approach to reachability analysis, in which a PEG is generated, and then a "local" reachability analysis is performed from each PEG node in parallel, ignoring host events. Other applications of PEGs are also given.

ACKNOWLEDGEMENTS

I would like to thank Professor Ural, of the University of Ottawa, for his assistance during the preparation of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 Protocol Validation

A *protocol* can be defined as a network of communicating processes and their associated input queues. One of the earliest and simplest protocol models is the *communicating finite state machine* (*CFSM*) model, in which each process is a CFSM [Zafi 78, Zafi 80, Rubi 82, Bran 83, Okum 87, Yuan 88, Holz 91]. The study of protocol validation is the study of decision problems based on protocols. A decision problem is a problem for which a solution, given a problem instance, consists of either "yes" or "no" [Gare 79]. In protocol validation, a problem instance is a protocol specification.

In a decision problem, the yes/no answer can be viewed as revealing whether or not the problem instance has a particular property. Protocol validation techniques can therefore be classified according to the properties that they are able to detect. Traditionally, protocol validation techniques have focused on the following properties:

- *deadlock* : "when each and every process has no alternative but to remain indefinitely in the same state" [Zafi 80]

- *unspecified reception* : "when a reception that can take place is not specified in the design" [Zafi 80]

- *unexecutable interactions* : "when a design includes message transmissions and receptions that cannot occur under normal operating conditions" [Zafi 80]

- *stable-state pairs* : "when a state x in one process and a state y in the other can be reached with both channels empty" [Zafi 80]

- *state ambiguity* : "when a state in one process can co-exist stably with several different states in the other process" [Zafi 80]

A property of wide interest that was not succinctly defined in [Zafi 80] is *channel overflow*. It may be defined as holding "when the number of messages on an input queue exceeds some prescribed bound".

With the exception of unexecutable interactions, these properties are defined in terms of *global states*, i.e. states of the protocol, consisting of the state of each process and the

contents of each input queue. The presence of unexecutable interactions is an event-oriented property, i.e. the assertion that some specified event cannot be executed. Techniques that focus on the listed properties, or some subset of these properties, include duologue-matrix analysis [Zafi 78]; reachability analysis [Zafi 80]; fair progress [Rubi 82]; maximal progress [Goud 84]; random search [West 86]; probabilistic verification [Maxe 87]; and scatter search [Holz 87].

Another property considered in some papers is *livelock*, or *tempo-blocking*, which can be defined as follows:

> "A livelock is a situation in which several processes keep exchanging messages, but without any effective work being accomplished" [Yuan 88].

Unlike the previous properties, the definition of livelock depends on the semantics of the protocol, i.e. the definition of *effective work*. A simple approach is to declare some transitions as representing progress, and define livelock as a cycle that contains no progress transitions.

Another approach has been to detect the presence or absence of properties specified by temporal logic formulae [Clar 86]. This approach also focuses on global states, since the temporal logic formulae are defined in terms of sequences of global states.

Two of the properties described above, unexecutable transitions and livelock, are defined in terms of events rather than global states. Many other properties could be defined in terms of sequences of executable events. This approach is discussed in the following section.

1.2 Language-Based Analysis

Okumura analyzed the language structure of CFSM-based protocols [Okum 87], but did not consider validation techniques. Languages were classified as *protocol languages* (i.e. executable global event sequences) or *process languages* (i.e. executable event sequences of an individual CFSM). Special consideration was also given to *null-accepted* languages, i.e. sets of executable sequences that end in a global state with all input queues empty. Null-accepted sequences correspond to stable-state pairs in the two-process case, or *stable state sets* (i.e. global states in which all input queues are empty) in the general case.

For *bounded protocols* (i.e. protocols in which the queue lengths do not exceed some finite bound), the protocol and process languages are regular [Hopc 79, Okum 87]. For *unbounded protocols*, the language structures are more complex. Okumura shows that, in the unbounded case, protocol languages are context-sensitive [Hopc 79]. In a protocol with more than two CFSMs, Okumura shows that a CFSM generates a phrase-structure grammar [Hopc 79], i.e. it has the computing power of a Turing machine [Hopc 79]. In protocols with exactly two CFSMs, it is shown that a process language (a "2-CFSM language") is context-sensitive, covers regular languages and overlaps but does not cover context-free languages [Hopc 79]. That is, every regular language is a 2-CFSM language, some context-free languages are 2-CFSM languages, and some 2-CFSM languages are context-sensitive but not context-free.

Okumura calls a process "effective" if all its specified sequences are executable, and calls a protocol "effective" if every process is effective [Okum 87]. A protocol or process which is not effective could be called "ineffective". Ineffectiveness corresponds to the existence of unexecutable sequences, which generalizes the concept of unexecutable transitions. That is, the existence of unexecutable transitions implies the existence of unexecutable sequences, but not vice versa. If the protocol designers intended a process to be effective, then ineffectiveness would be viewed as an undesirable property. Rudie and Wonham, however, present an approach to protocol synthesis in which one process plays the role of "supervisor", to restrict the executable sequences of another (the "plant") [Rudi 90]. With this approach, the plant would not usually be expected to be effective, but the supervisor might be.

In this thesis, we will develop a protocol validation technique to determine whether a process in a protocol is effective. Our first validation step will be to generate a *finite state machine* (*FSM*) that accepts the process language. We will call such an FSM a *process event graph* (*PEG*), and the process for which a PEG is generated will be called the *host process*. Each edge of the PEG will be labelled by some *host event*. The process language is then defined as if every PEG node were a final state. The second step in determining effectiveness is to compare the PEG with the specification graph, to determine whether they accept the same language. Although our main purpose is to determine effectiveness, a PEG could also be used for a number of other purposes, such as:

- identify particular specified sequences that are not executable

- determine whether particular specified sequences are executable

- generate local scenarios, i.e. executable sequences involving the host process only

In the PEGs generated by the algorithm developed in this thesis, every PEG node is mapped to a global state. This mapping permits the following application of a PEG:

- mark nodes of the PEG that map to global states from which there is a sequence of non-host events that leads to either a deadlock or a cycle of non-host events. Such nodes correspond to possible *process blockage*, i.e. global states from which the host process may be unable to progress, depending on events in the non-host processes.

An interesting property of every PEG, as generated by the algorithm developed in this thesis, is that, when the PEG generation does not detect error states, every reachable global state (i.e. every global state that can be reached by some execution sequence) is reachable by a sequence of non-host events from the global state mapped to some PEG node. This property permits us to use PEGs to introduce parallelism to reachability analysis, by the following two-step procedure:

1) generate a PEG for some process in a protocol

2) given n PEG nodes, perform n partial reachability analyses in parallel. For each PEG node, use the global state mapped to it as the initial global state in one of the partial reachability analyses. In each partial reachability analysis, consider non-host events only.

For maximum parallelism, each partial reachability analysis would begin as soon as the corresponding PEG node was generated, rather than waiting for the complete PEG to be generated.

In the following section, we outline the main objectives of the thesis.

1.3 Objectives

Our main objective is to develop an algorithm that efficiently determines whether a process in a protocol is effective. The approach we will take is to generate a PEG, and then

compare it with the process specification to determine whether there are any specified but unexecutable sequences. A secondary objective is to examine other potential uses of PEGs.

1.4 Organization of the Thesis

Chapter 2 ("Preliminaries") formally describes the CFSM model, presents an algorithmic definition of reachability analysis, and proves properties of the algorithm. It also introduces the *reachability graph*, and shows how reachability analysis can be used to generate a reachability graph.

Chapter 3 ("Process Event Graphs") defines the PEG, and develops algorithms to generate a PEG given a protocol with a specified host process. It begins with a tabular method that derives a PEG from a reachability graph, and then presents a series of improvements that lead to an algorithm that does not require a reachability graph. The improved algorithm traverses only part of the reachable global state space, and produces a smaller PEG than is produced by the tabular method.

Chapter 4 ("Applications of Process Event Graphs") examines three major applications of PEGs:

1) validating process effectiveness, i.e. determining whether all specified event sequences of the host process are executable

2) introducing parallelism to reachability analysis

3) detecting process blockage

Chapter 5 ("Related Algorithms") presents two algorithms (the maximal progress algorithm [Goud 84], and the YK-algorithm [Yuan 89]) that bear some resemblance to the PEG-generating algorithm developed in this thesis, and shows how these two algorithms differ from the PEG-generating algorithm. Neither the maximal progress algorithm nor the YK-algorithm was developed for the purpose of generating a PEG, but both employ the concept of a host process, and traverse part of the reachable global state space.

Chapter 6 presents conclusions and areas for future research. It is followed by a glossary of symbols and acronyms, and a bibliography.

2. PRELIMINARIES

2.1 The CFSM Model

Definition 2.1: A *protocol*, X, is a 4-tuple (<P>, <Q>, M, E) where

<P> is an n-tuple (P_1, ..., P_n) of processes, or CFSMs

<Q> is an n-tuple (Q_1, ..., Q_n) of perfect FIFO input queues for P_1, ..., P_n respectively

M is a message set

E is an event set

Messages in M are atomic. The reception of a message $m \in M$ is represented by +m and called a *receive event*. The transmission of a message $m \in M$ is represented by -m and called a *send event*. Events are defined formally as follows:

$E \equiv +E \cup -E$, where

$+E \equiv \{+m \mid m \in M\}$

$-E \equiv \{-m \mid m \in M\}$

+E is the set of receive events; -E is the set of send events.

A process P_i in protocol X is a 3-tuple (σ_i, δ_i, σ_{i0}) where

1) σ_i is a non-empty finite set of states, $\{\sigma_{i0}, ..., \sigma_{im}\}$

2) δ_i, the transition function, is a partial function from $\sigma_i \times E$ to σ_i

3) σ_{i0}, the initial state, is a designated element of σ_i

Q_i is the input queue for P_i. The role of the input queue is described in Definitions 2.5 and 2.6 below.

Definition 2.2: E_i, the event set associated with P_i, is defined as follows:

$E_i \equiv +E_i \cup -E_i$, where

$+E_i \equiv \{e \mid e \in +E, \delta_i(\sigma, e)$ is defined for some $\sigma \in \sigma_i\}$

$-E_i \equiv \{e \mid e \in -E, \delta_i(\sigma, e)$ is defined for some $\sigma \in \sigma_i\}$

$+E_i$ is the set of specified receive events for P_i; $-E_i$ is the set of specified send events for P_i.

Definition 2.3: M_i, the message set associated with P_i, is defined as follows:

$M_i \equiv +M_i \cup -M_i$

$+M_i \equiv \{m \mid +m \in +E_i\}$

$-M_i \equiv \{m \mid -m \in -E_i\}$

$+M_i$ is the set of messages specified as *receivable* by P_i; $-M_i$ is the set of messages specified as *sendable* by P_i. Note that messages specified as receivable or sendable are not necessarily received or sent in any executable sequence. Also, there may be messages sent to P_i that are not specified as receivable by P_i.

As a further restriction on the form of a protocol X, we require

$+M_i \cap +M_j = \varnothing \ \forall\ i, j, i \neq j, i \in [1,n], j \in [1,n]$

$-M_i \cap -M_j = \varnothing \ \forall\ i, j, i \neq j, i \in [1,n], j \in [1,n]$

$-M_i \cap +M_i = \varnothing \ \forall\ i, i \in [1,n]$

Definition 2.4: Let $P_i = (\sigma_i, \delta_i, \sigma_{i0})$ be a process, and let E_i^* denote the set of all finite or infinite sequences constructed from members of E_i. Function δ_i^*, the iterated operation of δ_i, is defined as follows:

1) $\delta_i^*(\sigma,\lambda) = \sigma$ for null sequence λ (i.e. λ is the trivial event sequence containing no events)

2) for $\alpha \in E_i^*$ and $e \in E_i$, where α is of finite length, $\delta_i^*(\sigma,\alpha\cdot e) = \delta_i(\delta_i^*(\sigma,\alpha),e)$,where $\alpha\cdot e$ represents the concatenation of α and e

3) when α is of infinite length, there can be no state σ' such that $\delta_i^*(\sigma,\alpha) = \sigma'$. However, if $\delta_i^*(\sigma,p)$ is defined for every finite prefix p of α, then we say that $\delta_i^*(\sigma,\alpha)$ is defined.

$\delta_i^*(\sigma_{i0},\alpha)$ is abbreviated to $\delta_i^*(\alpha)$. When $\delta_i^*(\alpha)$ is defined, we say α is a *specified sequence* of the process.

Definition 2.5: A *global state*, V, of protocol X is a 2n-tuple $V = (\langle v_i \rangle, \langle q_i \rangle)$, where v_i is the current state of P_i and q_i is a sequence of messages representing the contents of Q_i, the input queue to P_i. That is,

1) $v_i \in \sigma_i$

2) $q_i \in M^*$

The notation M^* denotes the set of all finite or infinite sequences constructed from members of M.

Definition 2.6: A *global state transition function* is a partial function, D, defined as follows, where V and V′ are global states:

1) $D(V, -m) = V'$, $m \in M$, iff $\exists\, i \in [1,n], j \in [1,n]$ such that

   a) $v'_i = \delta_i(v_i, -m)$, $v'_k = v_k \; \forall \; k \in [1,n], k \neq i$

   b) $q'_j = q_j \cdot m$, $q'_k = q_k \; \forall \; k \in [1,n], k \neq j$

2) $D(V, +m) = V'$, $m \in M$, iff $\exists\, j \in [1,n]$ such that

   a) $v'_j = \delta_j(v_j, +m)$, $v'_k = v_k \; \forall \; k \in [1,n], k \neq j$

   b) $m \cdot q'_j = q_j$, $q'_k = q_k \; \forall \; k \in [1,n], k \neq j$

The form of a global state transition is also restricted by Definition 2.3 above, which prevents a process from sending to itself, and prevents distinct processes from sending the same message, or receiving the same message.

Definition 2.7: D*, the iterated operation of D, is defined as follows:

1) $D^*(V, \lambda) = V$ for null sequence λ

2) for $\alpha \in E^*$ and $e \in E$, where α is of finite length, $D^*(V,\alpha \cdot e) = D(D^*(V,\alpha),e)$

3) when α is of infinite length, there can be no state V' such that $D^*(V,\alpha) = V'$. However, if $D^*(V,p)$ is defined for every finite prefix p of α, then we say that $D^*(V,\alpha)$ is defined.

$D^*(\mathbb{R}, \alpha)$ is written as $D^*(\alpha)$, where $\mathbb{R} = (\sigma_{10}, ..., \sigma_{n0}, \lambda, ..., \lambda)$ is the *initial global state* of protocol X. When $D^*(\alpha)$ is defined, we say α is an *executable sequence* of protocol X. For α of finite length, when $D^*(\alpha) = V$ is defined, V is said to be a *reachable global state*. The set of all reachable global states of protocol X is called the *reachable global state space* of X, denoted R(X). With unbounded queues, |R(X)| may be infinite. Also, since $D^*(\lambda) = \mathbb{R}$, $\mathbb{R} \in R(X)$.

Definition 2.8: A labelled digraph G is said to be a *process specification graph* for process P_i iff:

1) there is a one-to-one mapping between the nodes of G and the states of P_i

2) for every pair of nodes w_j, w_k of G there is an edge labelled e from w_j to w_k iff $\delta_i(v_j, e) = v_k$, where v_j and v_k are the states of P_i mapped to w_j and w_k respectively

For convenience, the nodes of a process specification graph will be identified with the corresponding process states, and the edges will be identified with the corresponding events.

Definition 2.9: A labelled digraph G is said to be a *reachability graph* for protocol X iff:

1) there is a one-to-one mapping between the nodes of G and the reachable global states of X

2) for every pair of nodes w_j, w_k of G there is an edge labelled e from w_j to w_k iff $D(V_j, e) = V_k$, where V_j and V_k are the reachable global states of X mapped to w_j and w_k respectively

For convenience, the nodes of a reachability graph will be identified with the corresponding global states, and the edges will be identified with the corresponding events.

Theorem 2.1: The reachability graph G for a protocol X is unique up to a renaming of the nodes.

Proof: Suppose there are two reachability graphs, G and G′, for X. Consider breadth-first traversals of G and G′, beginning at $\mathbb{R}$ in each case. Call $\mathbb{R}$ level 0, call its offspring level 1, and so on. Suppose the two traversals are identical up to level n. Then by definition of the reachability graph, they will be identical up to level n+1. Since the traversals are identical up to level 0, they are identical, by induction. But if breadth-first traversals of G and G′ are identical, G and G′ must be identical.

QED.

Definition 2.10: As justified by Theorem 2.1, the unique reachability graph of a protocol X will be denoted by G(X).

Definition 2.11: For $V = (v_1, ..., v_n, q_1, ..., q_n) \in R(X)$, and where B is a positive integer (the "queue bound"), error_state(V, B) is a Boolean function defined as follows:

error_state(V, B) = overflow(V, B) or unspec_rec(V) or deadlock(V)

overflow(V, B) = TRUE iff $\exists\, i \in [1, n]$ such that $|q_i| > B$

unspec_rec(V) = TRUE iff $\exists\, i \in [1, n]$ such that $q_i = m \cdot q'_i$ for some m and q'_i, and $\delta_i(v_i, +m)$ is not defined

deadlock(V) = TRUE iff $\forall\, i \in [1, n]$,

1) there is no m such that $\delta_i(v_i, -m)$ is defined, and

2) either

   a) $q_i = \lambda$, or

   b) $q_i = m \cdot q'_i$ for some m and q'_i, and $\delta_i(v_i, +m)$ is not defined

Definition 2.12: $R_B(X) \equiv \{ V = (v_1, ..., v_n, q_1, ..., q_n) \mid V \in R(X), |q_i| \le B \;\forall\, i \in [1, n]\}$

That is, $R_B(X)$ is the subset of the reachable global state space consisting of those global states in which no queue length exceeds B. Obviously $|R_B(X)|$ is finite.

2.2 Reachability Analysis

The basic idea behind reachability analysis is to traverse the reachable global state space R(X) of a protocol X, and examine each reachable global state for potential errors. Errors normally sought include deadlock, unspecified reception, and queue overflow. Unexecutable interactions, stable states, state ambiguities, and livelock are also of wide interest, as mentioned in the introduction, but they will not be considered here. Reachability analysis can also be used to generate a *reachability graph* (*RG*), which can be used for other purposes, such as analysis of temporal properties [Clar 86].

Consider the following algorithm (adapted from [Holz 91]), which we will call Algorithm 1:

```
analyze(X, B)
1.    R := (σ10, ..., σn0, λ, ..., λ); /* initial global state */
2.    W := {R};              /* working set of global states to analyze */
3.    A := ∅;                /* set of global states already analyzed */
4.    error := FALSE;
5.    while ((W ≠ ∅) and not error)
6.    {     V := a member of W;
7.          W := W - V;
8.          A := A ∪ {V};
9.          if (error_state(V, B))
10.         {     error := TRUE;
            }
11.         else
12.         {     for every event e such that D(V, e) is defined
13.               {     V' := D(V, e);
14.                     create edge labelled e from V to V';
15.                     if ((V' ∉ A) and (V' ∉ W))
16.                     {     W := W ∪ {V'};
                        }
                }
            }
      }
```

Algorithm 1: Reachability Analysis With RG Generation

Theorem 2.2: For any subset A of R(X), $|A| > |R_B(X)|$ implies $\exists\ V \in A$ such that overflow(V,B).

Proof: $|A| > |R_B(X)|$ implies $\exists\ V \in A$ such that $V \notin R_B(X)$. But for any global state $V = (v_1, ..., v_n, q_1, ..., q_n) \in R(X)$, if $V \notin R_B(X)$, then $\exists$ i such that $|q_i| > B$. Therefore, overflow(V,B).

QED.

Theorem 2.3: Algorithm 1 will eventually terminate for any input protocol X and positive bound B.

Proof: Insert assertions (indicated by square brackets) into Algorithm 1 as follows :

```
analyze(X, B)
1.      R := (σ10, ..., σn0, λ, ..., λ);
2.      W := {R};
3.      A := ∅;
(1)     [W ∩ A = ∅]
(2)     [|A| = 0]
4.      error := FALSE;
5.      while ((W ≠ ∅) and not error)
        {
(3)     [W ∩ A = ∅]
(4)     [let k = |A|]
6.              V := a member of W;
7.              W := W - V;
(5)     [W ∩ A = ∅]
(6)     [V ∉ A]
8.              A := A ∪ {V};
(7)     [W ∩ A = ∅]
(8)     [|A| = k + 1]
9.              if (error_state(V, B))
10.             {       error := TRUE;
                }
11.             else
12.             {       for every event e such that D(V, e) is defined
13.                     {       V' := D(V, e);
14.                             create edge labelled e from V to V';
15.                             if ((V' ∉ A) and (V' ∉ W))
16.                             {       W := W ∪ {V'};
(9)     [W ∩ A = ∅]
                                }
                        }
                }
        }
```

Assertion (1) is clearly true. After each change to W and A it remains true, i.e. at assertions (5), (7), and (9). Therefore assertion (3) is true. From (3), we can conclude (6). From (6), we can conclude (8).

Assertion (2) is true due to statement 3. From assertion (2), |A| is 0 on entry to the loop (at statement 5). From assertion (8), |A| increases by 1 with each iteration through the loop.

So after $|R_B(X)| + 1$ iterations, $|A| = |R_B(X)| + 1$. From Theorem 2.2, there must then be some $V \in A$ such that overflow(V, B), and therefore error_state(V). In that case, when V is generated, error becomes true, and the loop terminates.

QED.

Theorem 2.4: On termination of Algorithm 1, the following assertion is true:

error iff error_state(V, B) for some $V \in R(X)$

Proof: Insert assertions into Algorithm 1 as follows:

```
analyze(X, B)
1.      P := (σ10, ...., σn0, λ, ...., λ);
2.      W := {P};
3.      A := ∅;
(1)     [V ∈ W ∪ A → V ∈ R(X)]
4.      error := FALSE;
(2)     [not  error]
(3)     [(not error) or (error and (error_state(V, B) for some V ∈  R(X)))]
5.      while ((W ≠ ∅) and not error)
        {
(4)     [V ∈ W ∪ A → V ∈ R(X)]
(5)     [not  error]
(6)     [(not error) or (error and (error_state(V, B) for some V ∈  R(X)))]
6.              V := a member of W;
(7)     [V ∈  R(X)]
7.              W := W - V;
8.              A := A ∪ {V};
(8)     [V ∈ W ∪ A → V ∈ R(X)]
9.              if  (error_state(V,  B))
10.             {       error := TRUE;
(9)     [error and (error_state(V, B) for some V ∈  R(X))]
                }
11.             else
                {
(10)    [not  error]
12.                     for every event e such that D(V, e) is defined
13.                     {       V' := D(V, e);
(11)    [V' ∈  R(X)]
14.                             create edge labelled e from V to V';
15.                             if ((V' ∉  A) and (V' ∈  W))
16.                             {       W := W ∪ {V'};
(12)    [V ∈ W ∪ A → V ∈ R(X)]
                                }
                        }
                }
(13)    [(not error) or (error and (error_state(V, B) for some V ∈  R(X)))]
        }
```

Assertions (1), (2) and (3) are clearly true. Therefore assertions (4), (5) and (6) are true on entry to the loop. Assertion (7) follows from statement 6 and assertion (4); (8) follows from statements 7 and 8 and assertions (7) and (4); (9) follows from statements 9 and 10, and assertion (7); (10) follows from assertion (5); (11) follows from statement 13 and assertion (7); (12) follows from statement 16 and assertions (8) and (11); (13) follows from assertions (9) and (10). Assertions (4) and (6) are maintained at assertions (12) and (13). Therefore assertion (6) is true on termination of the algorithm. But it can be rewritten as "error iff error_state(V, B) for some V ∈ R(X)".

QED.

Theorem 2.5: If error_state(V, B) is false for every $V \in R(X)$, then $A = R(X)$ on termination of Algorithm 1. That is, if there are no error states, Algorithm 1 can be said to *traverse* the reachable global state space of the given protocol.

Proof: Insert assertions into Algorithm 1 as follows:

```
analyze(X, B)
1.      R := (σ10, ..., σn0, λ, ..., λ);
2.      W := {R};
3.      A := ∅;
4.      error := FALSE;
(1)     [for every Vj ∈ A, for every e such that D(Vj, e) is defined,
         D(Vj, e) ∈ W ∪ A]
(2)     [W ∩ A = ∅]
(3)     [not error]
(4)     [R ∈ W ∪ A]
5.      while ((W ≠ ∅) and not error)
        {
(5)     [for every Vj ∈ A, for every e such that D(Vj, e) is defined,
         D(Vj, e) ∈ W ∪ A]
(6)     [W ∩ A = ∅]
(7)     [not error]
(8)     [R ∈ W ∪ A]
6.              V := a member of W;
7.              W := W - V;
8.              A := A ∪ {V};
(9)     [for every Vj ∈ A - {V}, for every e such that D(Vj, e) is defined,
         D(Vj, e) ∈ W ∪ {A - {V}}]
(10)    [W ∩ A := ∅]
(11)    [not error_state(V, B)]
(12)    [R ∈ W ∪ A]
9.              if (error_state(V, B))
10.             {       error := TRUE;
                }
11.             else
12.             {       for every event e such that D(V, e) is defined
13.                     {       V' := D(V, e);
14.                             create edge labelled e from V to V';
15.                             if ((V' ∉ A) and (V' ∉ W))
16.                             {       W := W ∪ {V'};
                                }
                        }
(13)    [for every e such that D(V, e) is defined, D(V, e) ∈ W ∪ A]
(14)    [for every Vj ∈ A, for every e such that D(Vj, e) is defined,
         D(Vj, e) ∈ W ∪ A]
(15)    [W ∩ A = ∅]
(16)    [not error]
(17)    [R ∈ W ∪ A]
                }
        }
```

Assertions (1) to (4) are clearly true, and establish the invariants at (5) to (8) respectively. Assertion (7) is maintained because of our assumption that error_state(V, B) is false for all $V \in R(X)$. Assertions (5), (6) and (8) are maintained as shown by assertions (9) to (17). By Theorem 2.3, the algorithm eventually terminates, and by assertion (16) and statement 5, $W = \emptyset$ on termination. Therefore, by assertion (17), $\mathbb{R} \in A$ at termination.

Therefore, by assertion (14), every state reachable by one step from $\mathbb{R}$ is in A. By repeated application of (14), every state reachable from $\mathbb{R}$ is in A.

QED.

Theorem 2.6: If error_state(V, B) is false for every $V \in R(X)$, then the graph generated by Algorithm 1, consisting of the nodes in A and the edges created at statement 14, is G(X).

Proof: From Theorem 2.5, A = R(X) on termination of the algorithm. But states are added to A at statement 8 only. For each state added at statement 8, all its outgoing edges are generated at statement 14, and no other edges are created.

QED.

Algorithm 1 terminates when an error state is encountered. In some cases, it might be preferable to mark the error state, make it a terminal node, and continue processing. This can be achieved by a simple modification to Algorithm 1, as follows:

```
analyze(X, B)
1.      ℝ := (σ10, ..., σn0, λ, ..., λ); /* initial global state */
2.      W := {ℝ};              /* working set of global states to analyze */
3.      A := ∅;                /* set of global states already analyzed */
4.      while (W ≠ ∅)
5.      {       V := a member of W;
6.              W := W - V;
7.              A := A ∪ {V};
8.              if (error_state(V, B))
9.              {       error(V) := TRUE;
                }
10.             else
11.             {       error(V) := FALSE;
12.                     for every event e such that D(V, e) is defined
13.                     {       V' := D(V, e);
14.                             create edge labelled e from V to V';
15.                             if ((V' ∉ A) and (V' ∉ W))
16.                             {       W := W ∪ {V'};
                                }
                        }
                }
        }
```

Algorithm 2: Reachability Analysis Without Terminating on Errors

The proofs applied to Algorithm 1 can be applied, with minor variations, to Algorithm 2. Proof of termination is more complicated, but can be derived from the fact that W never contains a node with a queue length greater than B, and a node never appears in A or W more than once. Therefore only a finite number of nodes are generated, and the algorithm eventually terminates.

Henceforth, all references to RGs assume that the RG was generated by Algorithm 2, not Algorithm 1. That is, if error_state(V,B), then V is a terminal node.

In the following chapter, we define the process event graph (PEG) and develop algorithms that generate a PEG, given a protocol and specified host process. Two approaches are considered:

1) generating a PEG by first generating an RG

2) generating a PEG without first generating an RG

3. PROCESS EVENT GRAPHS

3.1 Definitions

Definition 3.1: The restriction of event sequence α to event set E_i, denoted $\alpha{:}E_i$, is defined as follows:

1) $\lambda{:}E_i = \lambda$

2) $(\alpha{\cdot}e){:}E_i = (\alpha{:}E_i){\cdot}e$ if $e \in E_i$

3) $(\alpha{\cdot}e){:}E_i = \alpha{:}E_i$ if $e \notin E_i$

Definition 3.2: The *protocol language* for protocol X with bound B, denoted L(X,B), is defined as follows:

$L(X,B) = \{\alpha \mid D^*(\alpha)$ is defined, and there is no prefix p of α such that $|p| < |\alpha|$ and error_state$(D^*(p),B)$ = TRUE$\}$

Definition 3.3: The *process language* for process P_i in protocol X with bound B, denoted $L_i(X,B)$, is defined as follows:

$L_i(X,B) = \{\beta \mid \exists\ \alpha \in L(X,B),\ \beta = \alpha{:}E_i\}$

Definition 3.4: A labelled digraph G is said to be a *process event graph* (*PEG*) for process P_i iff there is a node r (the "initial node") of G such that the set of directed sequences beginning at r in G is identical to $L_i(X,B)$. That is, a PEG is an FSM that accepts $L_i(X,B)$, where every PEG node is treated as a final state.

Definition 3.5: An *ε-move* of an FSM is a spontaneous state transition, i.e. a transition with no input.

Definition 3.6: An *ε-sequence* is a (possibly null) sequence that contains only ε-moves.

Definition 3.7: An *ε-cycle* is a non-null ε-sequence that contains a cycle.

The remainder of Chapter 3 presents algorithms that generate PEGs. The algorithms do not generate identical PEGs, although all generated PEGs conform to Definition 3.4. When it is necessary to distinguish between PEGs generated by the different algorithms, the following terms will be used:

1) PEG_T (PEG generated by tabular method)

2) PEG_3 (PEG generated by Algorithm 3)

3) PEG_4 (PEG generated by Algorithm 4)

4) PEG_5 (PEG generated by Algorithm 5)

3.2 Generating PEGs from Reachability Graphs

Taking the perspective of classical language theory [Hopc 79], we could view the RG as an FSM, and view all non-host edges (i.e. transitions executed by processes other than the host process) as ε-moves. If every global state were treated as a final state, the RG with ε-moves would accept $L_i(X,B)$. It is shown in [Hopc 79] that FSMs with ε-moves can be converted to equivalent FSMs without ε-moves. This approach can be used to generate a PEG from an RG. In [Hopc 79], a tabular method is used.

The tabular method can be described informally as follows:

- create a state/event table with one row per global state, one column per host event, and an extra column for ε
- in every state/event table entry, excluding the ε column, list the states that are reachable from the corresponding state by some sequence consisting of an ε-sequence, followed by a single host event corresponding to the column, followed by an ε-sequence
- in every row of the ε column, list the states that are reachable from the corresponding state by an ε-sequence
- construct a graph from the table, creating a host edge from every state to the states specified in its row, excluding the ε column, and label each edge with the event of the corresponding column
- mark final states as follows:
  - every state that was final in the original graph (i.e. with ε-moves) is also final in the resultant graph (i.e. without ε-moves)

- mark a state as final if there is an ε-sequence from it to a final state (there is an ε-sequence from s_1 to s_2 iff s_2 appears in the ε column of the s_1 row)

To illustrate the tabular method, consider the following protocol, which we will call Protocol 1:

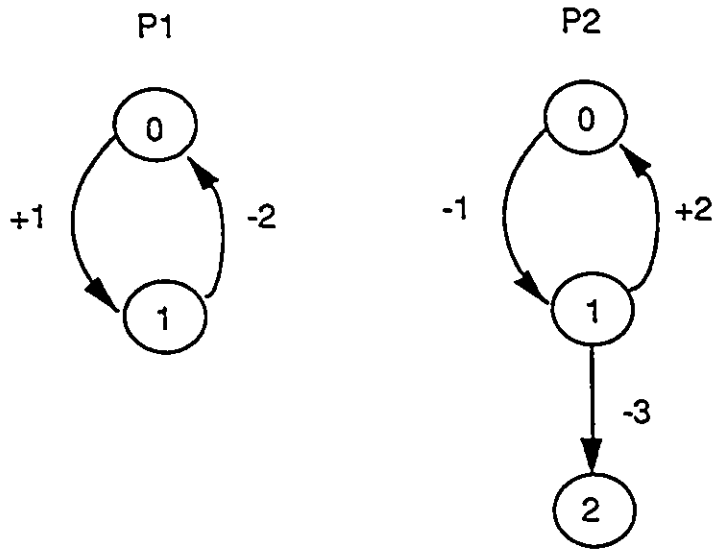

Figure 3.1 - Protocol 1

The RG for Protocol 1 is shown in Figure 3.2:

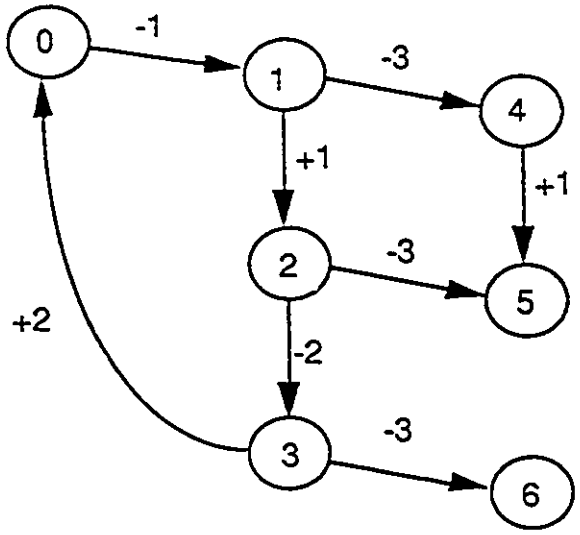

Figure 3.2 - Reachability Graph for Protocol 1

To generate a PEG, every state is marked as final. Therefore every state in the output graph will be marked as final, so there is no need to generate the ε column and compute the final states. With P_1 as the host process, the completed state/event table is as follows:

event / state	+1	-2
0	2,5	-
1	2,5	-
2	-	0,1,3,4,6
3	2,5	-
4	5	-
5	-	-
6	-	-

Table 3.1 - Result of Tabular Method for Protocol 1 with P_1 as Host

The graph derived from Table 3.1 is shown in Figure 3.3:

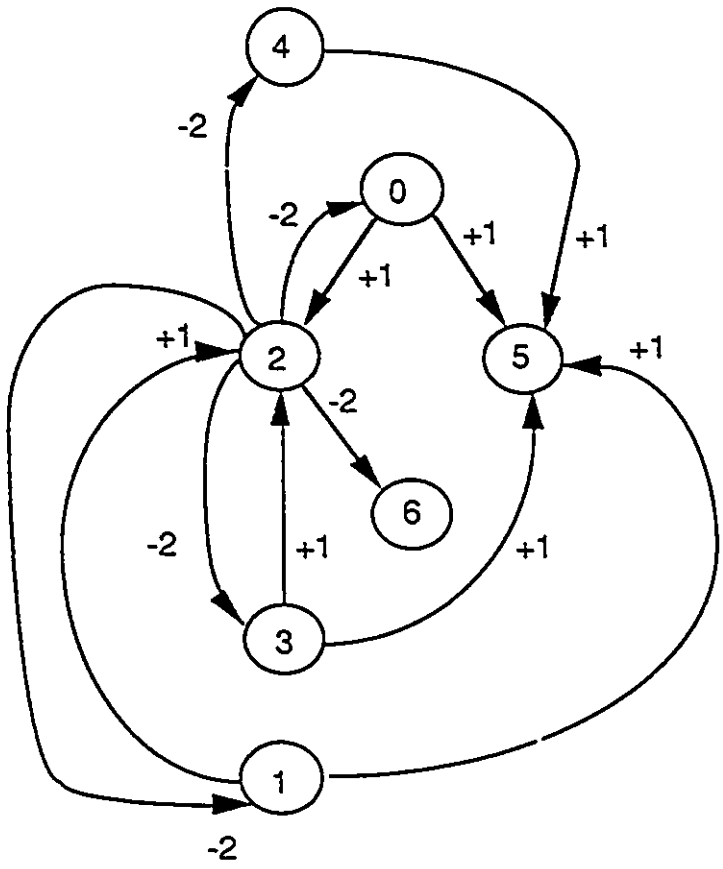

Figure 3.3 - Result of Tabular Method for Protocol 1 with P_1 as Host

As noted earlier, a PEG produced by the tabular method will be called a PEG_T, where the distinction is required. From Figure 3.3, which represents a PEG_T for P_1 in Protocol 1, it is apparent that the tabular method does not, in general, produce a minimal PEG. This can be seen by the fact that the specification graph for P_1 is, by inspection, a PEG.

The tabular method contains two main drawbacks:

1) an RG must be generated before it is performed

2) the output graph is far from minimal

The next section considers methods of generating PEGs directly, without first generating an RG.

3.3 Generating PEGs Directly

The tabular method can be modified to generate a PEG from the protocol specification, without requiring that an RG be generated first. The idea is to use a modified reachability analysis, and generate a PEG instead of generating an RG. This requires Theorem 3.1, which is presented below. The phrase "sequence of type $\varepsilon^*H\varepsilon^*$" refers to a sequence consisting of an ε-sequence, followed by a host transition, followed by an ε-sequence. The mapping between PEG_T nodes and global states referred to in Theorem 3.1 is the obvious one-to-one mapping induced by the tabular method, which generates a PEG_T node for each global state.

Theorem 3.1: The PEG_T node mapped to a global state V is reachable from the PEG_T node mapped to $\mathbb{R}$ iff V is reachable from $\mathbb{R}$ by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$.

Proof: The proof will be divided into two cases, and PEG_T nodes will be identified with the corresponding global states:

1) every V that is reachable from $\mathbb{R}$ in the PEG_T is reachable from $\mathbb{R}$ by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$ in the RG

2) every V that is reachable from $\mathbb{R}$ by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$ in the RG is reachable from $\mathbb{R}$ in the PEG_T

For part (1), consider any V that is reachable from $\mathbb{R}$ in the PEG_T. Let the sequence from $\mathbb{R}$ to V in the PEG_T be $\alpha = e_1e_2...e_n$, where e_i is an event of type H for $i \in [1,n]$. Since every e_i corresponds to a sequence of type $\varepsilon^*H\varepsilon^*$ in the RG, V is reachable from $\mathbb{R}$ by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$ in the RG.

For part (2), consider any V that is reachable from $\mathbb{R}$ by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$ in the RG. Clearly V is reachable from $\mathbb{R}$ in the PEG_T.

QED.

The tabular method generates a PEG edge from every global state from which there is a sequence of type $\varepsilon^*H\varepsilon^*$. From Theorem 3.1, there is no point in generating edges from states that are not reachable from $\mathbb{R}$ by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$. This allows us to generate a PEG during a modified reachability analysis, as shown in Algorithm 3:

```
genPEG(X, B, i)
1.      R := (σ10, ..., σn0, λ, ..., λ);/* initial global state */
2.      W := {R};               /* working set of global states to analyze */
3.      A := ∅;                 /* set of global states already analyzed */
4.      while (W ≠ ∅)
5.      {       V := a member of W;
6.              W := W - V;
7.              A := A ∪ {V};
8.              if (error_state(V, B))
9.              {       error(V) := TRUE;
                }
10.             else
11.             {       error(V) := FALSE;
12.                     for every executable sequence α of type ε*Hε*
                        beginning at V
13.                     {       V' := D*(V, α);
14.                             create edge labelled e from V to V',
                                where e is label of the H event;
15.                             if ((V' ∉ A) and (V' ∉ W))
16.                             {       W := W ∪ {V'};
                                }
                        }
                }
        }
```

Algorithm 3: PEG Generation During Reachability Analysis

Algorithm 3 duplicates the essential feature of the tabular method: wherever there is a sequence of type ε*Hε* from V to V', an edge representing the host transition is drawn from V to V'. Applied to Protocol 1, Algorithm 3 proceeds as follows:

1) Set A empty and store node 0 (i.e. $\mathbb{R}$) in W.

2) Remove node 0 from W and add it to A. There are 3 sequences of type ε*Hε* beginning at node 0:

   a) (-1,-3,+1) to node 5

   b) (-1,+1) to node 2

   c) (-1,+1,-3) to node 5

   So add nodes 2 and 5 to W, and draw an edge labelled +1 from node 0 to node 2, and two +1 edges from node 0 to node 5.

3) Remove node 2 from W and add it to A. There are 5 sequences of type ε*Hε* beginning at node 2:

   a) (-2) to node 3

   b) (-2,-3) to node 6

   c) (-2,+2) to node 0

   d) (-2,+2,-1) to node 1

   e) (-2,+2,-1,-3) to node 4

   Node 0 is already in A. So add nodes 1, 3, 4 and 6 to W, and draw an edge labelled -2 from node 2 to nodes 0, 1, 3, 4, and 6.

4) Remove node 5 from W and add it to A. Node 5 is an error state due to the unspecified reception of message 3, so no transitions are executed from node 5.

5) Remove node 1 from W and add it to A. There are 3 sequences of type ε*Hε* beginning at node 1:

   a) (-3,+1) to node 5

   b) (+1) to node 2

   c) (+1,-3) to node 5

   Nodes 2 and 5 are already in A, so just draw an edge labelled +1 from node 1 to node 2, and two +1 edges from node 1 to node 5.

6) Remove node 3 from W and add it to A. There are 3 sequences of type ε*Hε* beginning at node 3:

   a) (+2,-1,-3,+1) to node 5

   b) (+2,-1,+1) to node 2

   c) (+2,-1,+1,-3) to node 5

Nodes 2 and 5 are already in A, so just draw an edge labelled +1 from node 3 to node 2, and two +1 edges from node 3 to node 5.

7) Remove node 4 from W and add it to A. There is 1 sequence of type $\varepsilon^*H\varepsilon^*$ beginning at node 4:

   a) (+1) to node 5

   Node 5 is already in A, so just draw an edge labelled +1 from node 4 to node 5.

8) Remove node 6 from W and add it to A. Node 6 is a deadlock state, so error_state(node 6) is true. Now W is empty, and the algorithm terminates.

Algorithm 3 has one advantage over the tabular method: because no RG is generated, storage requirements may be lower (i.e. only PEG edges, not RG edges, are stored). However, Algorithm 3 has four apparent flaws:

1) redundant edges are introduced when multiple sequences of type $\varepsilon^*H\varepsilon^*$ from s_1 to s_2 have the same H event

2) from a given node V, every sequence of type $\varepsilon^*H\varepsilon^*$ is examined. This differs from the tabular method, which examines every other node to see if it is reachable by a sequence of type $\varepsilon^*H\varepsilon^*$ from V. The problem is that, if an ε-cycle exists, there may be an infinite number of sequences of type $\varepsilon^*H\varepsilon^*$ from V, in which case Algorithm 3 will not terminate.

3) intermediate global states traversed by a sequence of type $\varepsilon^*H\varepsilon^*$ are not examined for errors. For example, the presence of an unspecified reception in the ε^* prefix is not detected, so the sequence of type $\varepsilon^*H\varepsilon^*$ may not correspond to a sequence in the RG.

4) as with a PEG_T, a PEG_3 (i.e. a PEG generated by Algorithm 3) is apparently much larger than necessary

Problems 1 to 3 can be resolved as follows:

1) do not generate a new edge if an identical edge already exists

2) consider cycle-free sequences of type $\varepsilon^*H\varepsilon^*$ only

3) check for errors in the global states traversed by the sequences of type $\varepsilon^*H\varepsilon^*$, and terminate a sequence upon encountering an error state, as done by Algorithm 2. That is, consider error-free or error-terminated sequences of type $\varepsilon^*H\varepsilon^*$ only.

With these corrections (problem 4 will be addressed later), Algorithm 3 becomes Algorithm 4:

```
genPEG(X, B, i)
1.    R := (σ10, ..., σn0, λ, ..., λ);/* initial global state */
2.    W := {R};              /* working set of global states to analyze */
3.    A := ∅;                /* set of global states already analyzed */
4.    while (W ≠ ∅)
5.    {     V := a member of W;
6.          W := W - V;
7.          A := A ∪ {V};
8.          if (error_state(V, B))
9.          {     error(V) := TRUE;
            }
10.         else
11.         {     error(V) := FALSE;
12.               for every error-free or error-terminated, cycle-free executable
                  sequence α of type ε*Hε* beginning at V
13.               {     V' := D*(V, α);
14.                     if there is no edge labelled e from V to V',
                        where e is label of the H event;
15.                     {     create edge labelled e from V to V';
                        }
16.                     if ((V' ∈ A) and (V' ∈ W))
17.                     {     W := W ∪ {V'};
                        }

                  }
            }
      }
```

Algorithm 4: Algorithm 3 with Improvements

As noted above, a PEG generated by Algorithm 4 will be called a PEG_4, where the distinction is required. Applied to Protocol 1, Algorithm 4 generates the same PEG as the tabular method (Figure 3.3). However, Protocol 2 reveals a case in which the tabular method and Algorithm 4 generate different PEGs:

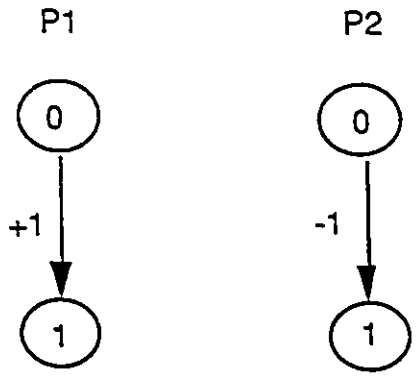

Figure 3.4 - Protocol 2

The RG for Protocol 2, and the PEGs produced by the tabular method and Algorithm 4 with P_1 as host are shown in Figure 3.5:

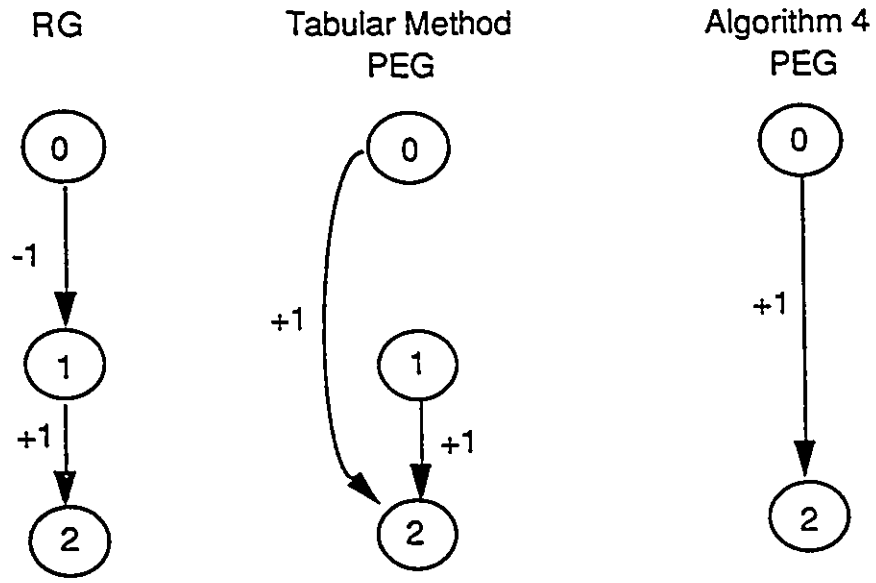

Figure 3.5 - Reachability Graph and PEGs for Protocol 2 with P_1 as Host

As can be seen from this example, the PEG_T may contain unreachable nodes, while the PEG_4 does not.

Although Algorithm 4 avoids the need to generate an RG, its output PEG is still apparently larger than necessary (Figure 3.3). We will now consider a series of theorems that lead to an improved algorithm, which generates much smaller PEGs. The central idea is that many of the RG sequences correspond to race conditions. That is, from a given global state there may be many sequences that lead to the same state, and differ only in the

global ordering of events, while retaining the same local order within each process. For purposes of PEG generation, we can avoid traversing the redundant sequences. We begin by dividing H and ε events into four categories:

Type	Event
H_S	send by host process
H_r	receive by host process
N_S	send to host process, by non-host process
N_O	other (i.e. receive by non-host, or send from non-host to non-host)

Table 3.2 - Event Types

When discussing the special case in which there are only two processes, N_O may be replaced by N_r, i.e. a receive by the non-host process.

These type identifiers can be used in regular expressions. An ε-sequence corresponds to a sequence of type $(N_S+N_O)^*$, i.e. the sequence of event types traversed can be expressed by the regular expression $(N_S+N_O)^*$. A sequence that contains some number of ε-moves and one host event can be expressed as $\varepsilon^*H\varepsilon^*$, or $(N_S+N_O)^*(H_S+H_r)(N_S+N_O)^*$.

Theorem 3.2: Given $V = (v_1, ..., v_n, q_1, ..., q_n) \in R(X)$, such that the host process is P_i and $q_i = \lambda$, and given any $\alpha \in E^*$ such that α is of type $H_S+(N_O{}^*N_SH_r)$, and $D^*(V,\alpha) = V' = (v_1', ..., v_n', q_1', ..., q_n')$, then $q_i' = \lambda$.

Proof: Either α is of type H_S or it is of type $N_O{}^*N_SH_r$. If it is of type H_S, i.e. a host send, then it does not affect the host input queue. A sequence of type $N_O{}^*N_S$ sequence adds one message to the host input queue, and an H_r event removes one message from the host input queue, so a sequence of type $N_O{}^*N_SH_r$ sequence does not change the length of the host input queue.

QED.

Theorem 3.3: Given $V = (v_1, ..., v_n, q_1, ..., q_n) \in R(X)$, such that the host process is P_i and $q_i = \lambda$, and given any $\alpha \in E^*$ such that α is of type $\varepsilon^*H\varepsilon^*$ and $D^*(V,\alpha)$ is defined, then $\exists\ \beta \in E^*$ such that β is of type $H_S+(N_O{}^*N_SH_r)$ and $D^*(V,\beta)$ is defined and $\beta{:}E_i = \alpha{:}E_i$.

Proof: Either α is of type $(N_S+N_O)^*H_S(N_S+N_O)^*$, or it is of type $(N_S+N_O)^*H_r(N_S+N_O)^*$. Suppose it is of type $(N_S+N_O)^*H_S(N_S+N_O)^*$. Sequences of type $(N_S+N_O)^*$ do not affect the host state, and the executability of H_S events does not depend on the contents of the host input queue. Therefore if the H_S event is executable after the $(N_S+N_O)^*$ prefix, it must be executable before the $(N_S+N_O)^*$ prefix. Therefore there must be an executable sequence β of type H_S such that the host event in β is identical to the host event in α.

Now suppose α is of type $(N_S+N_O)^*H_r(N_S+N_O)^*$. Since $q_i = \lambda$, there must be at least one N_S event in the $(N_S+N_O)^*$ prefix. That is, the type of α must be expressible as $N_O^*N_S(N_S+N_O)^*H_r(N_S+N_O)^*$. But since the sequence of type $N_O^*N_S(N_S+N_O)$ does not affect the host state, the H_r event must be executable after the $N_O^*N_S$ prefix. Therefore there must be an executable sequence β of type $N_O^*N_SH_r$ such that the host event in β is identical to the host event in α.

QED.

Definition 3.8: A protocol is said to be *mono-receiving* if no process receives messages from more than one other process.

Theorem 3.4: Given a mono-receiving protocol X, and given V, α, and β defined as in Theorem 3.3, where $D^*(\beta)$ is not an error state, there is an ε-sequence χ such that $D^*(V,\alpha) = D^*(V,\beta\cdot\chi)$.

Proof: As in Theorem 3.3, there are two cases to consider:

1) α is of type $(N_S+N_O)^*H_S(N_S+N_O)^*$ and β is of type H_S

2) α is of type $(N_S+N_O)^*H_r(N_S+N_O)^*$ and β is of type $N_O^*N_SH_r$

In case 1, consider α as the concatenation of three sequences, i.e. α = abc, where a is the $(N_S+N_O)^*$ prefix, b is the H_S event, and c is the $(N_S+N_O)^*$ suffix. Then β = b. Because X is mono-receiving, and $D^*(b)$ is not an error state, b does not affect the executability of a. Therefore bac must be an executable sequence. Then $D^*(V,\alpha) = D^*(V,abc) = D^*(V,bac) = D^*(D^*(V,b), ac) = D^*(D^*(V,\beta),ac) = D^*(V,\beta\cdot ac)$. Therefore $D^*(V,\alpha) = D^*(V,\beta\cdot ac)$, and ac is of type $(N_S+N_O)^*$.

In case 2, Theorem 3.3 shows that α is expressible as $N_O^*N_S(N_S+N_O)^*H_r(N_S+N_O)^*$. Define α = abcd, where a is of type $N_O^*N_S$, b is of type $(N_S+N_O)^*$, c is of type H_r, and d

is of type $(N_S+N_O)^*$. Then β = ac. Since c does not affect the executability of b, the sequence acbd must be executable. Then $D^*(V,\alpha) = D^*(V,abcd) = D^*(V,acbd) = D^*(D^*(V,ac),bd) = D^*(D^*(V,\beta),bd) = D^*(V,\beta\cdot bd)$. Therefore $D^*(V,\alpha) = D^*(V,\beta\cdot bd)$, and bd is of type $(N_S+N_O)^*$.

QED.

The following graph summarizes the relationship between V, α, β, and χ in Theorem 3.4:

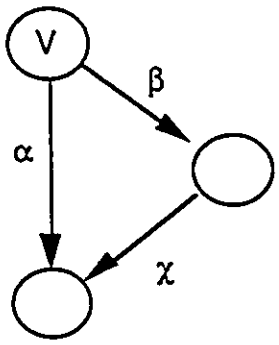

Figure 3.6 - Relationships of Objects in Theorem 3.4

The two cases that Figure 3.6 represents are as follows:

1) α is of type $(N_S+N_O)^*H_S(N_S+N_O)^*$

   β is of type H_S

   χ is of type $(N_S+N_O)^*$

2) α is of type $(N_S+N_O)^*H_r(N_S+N_O)^*$

   β is of type $N_O{}^*N_SH_r$

   χ is of type $(N_S+N_O)^*$

Theorem 3.5: Given X, V, α, β and χ defined as in Theorem 3.4, if there is a sequence α' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(V,\alpha\cdot\alpha')$ is defined, then there is a sequence β' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(V,\beta\cdot\beta')$ is defined, and $\alpha{:}E_i = \beta{:}E_i$, and $D^*(V,\alpha\cdot\alpha') = D^*(V,\beta\cdot\beta')$.

Proof: From Theorem 3.4, we know there is an ε-sequence χ such that $D^*(V,\beta\cdot\chi) = D^*(V,\alpha)$. Let $\beta' = \chi\cdot\alpha'$. Then β' is of type $\varepsilon^*H\varepsilon^*$, and $D^*(V,\beta\cdot\beta') = D^*(V,\beta\cdot\chi\cdot\alpha') = D^*(D^*(V,\beta\cdot\chi),\alpha') = D^*(D^*(V,\alpha),\alpha') = D^*(V,\alpha\cdot\alpha')$.

QED.

Theorem 3.6: For every sequence α of type $(H_S+(N_O{}^*N_SH_r))^*$ such that $D^*(\alpha)$ is defined, $q_i = \lambda$ at $D^*(\alpha)$.

Proof: (by induction on $|\alpha{:}E_i|$).

Basis ($|\alpha{:}E_i| = 1$): By Theorem 3.2.

Induction Step: Suppose Theorem 3.6 is true for all α such that $|\alpha{:}E_i| = n$. Consider any α' of type $H_S+(N_O{}^*N_SH_r)$ such that $D^*(\alpha\cdot\alpha')$ is defined. By the induction hypothesis, $q_i = \lambda$ at $D^*(\alpha)$. So by Theorem 3.2, $q_i = \lambda$ at $D^*(\alpha\cdot\alpha')$.

QED.

Figure 3.7 illustrates the relationships among objects that will be referenced in the proof of Theorem 3.7:

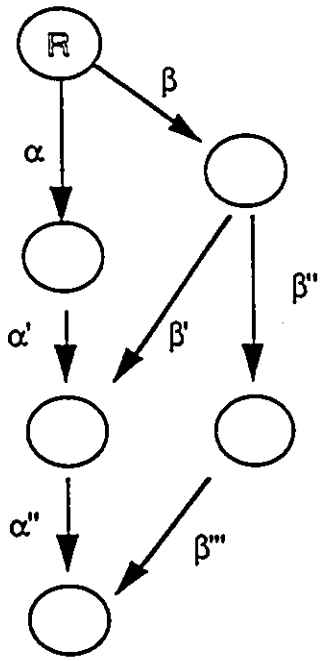

Figure 3.7 - Relationships of Objects in Theorem 3.7

Theorem 3.7: Given a mono-receiving protocol X, for every sequence α of type $(\varepsilon^*H\varepsilon^*)^*$ such that $D^*(\alpha)$ is defined, there is a sequence β of type $(H_S+(N_O{}^*N_SH_r))^*$ such that $D^*(\beta)$ is defined and $\alpha{:}E_i = \beta{:}E_i$ and the following is true:

> for every sequence α' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\alpha{\cdot}\alpha')$ is defined, there is a sequence β' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\beta{\cdot}\beta')$ is defined and $\alpha'{:}E_i = \beta'{:}E_i$ and $D^*(\alpha{\cdot}\alpha') = D^*(\beta{\cdot}\beta')$

Proof: (by induction on $|\alpha{:}E_i|$)

Basis ($|\alpha{:}E_i| = 1$): By Theorems 3.3 and 3.5.

Induction Step: Suppose Theorem 3.7 is true for all α such that $|\alpha{:}E_i| = n$. Consider any sequence $\alpha{\cdot}\alpha'$ of type $(\varepsilon^*H\varepsilon^*)^*$ such that $|(\alpha{\cdot}\alpha'){:}E_i| = n+1$, where $|\alpha{:}E_i| = n$, and $|\alpha'{:}E_i| = 1$. Therefore α' is of type $\varepsilon^*H\varepsilon^*$. Consider the β that corresponds to α in the inductive hypothesis. By the inductive hypothesis, there is a sequence β' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\beta{\cdot}\beta')$ is defined, and $\alpha'{:}E_i = \beta'{:}E_i$ and $D^*(\alpha{\cdot}\alpha') = D^*(\beta{\cdot}\beta')$. By Theorem 3.6, $q_i = \lambda$ at $D^*(\beta)$. By Theorem 3.3, there is a sequence β'' of type $H_S+(N_O{}^*N_SH_r)$ such that $D^*(\beta{\cdot}\beta'')$ is defined and $\beta''{:}E_i = \beta'{:}E_i$. Consider any sequence α'' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\beta{\cdot}\beta'{\cdot}\alpha'')$ is defined. By Theorem 3.5, there is a sequence β''' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\beta{\cdot}\beta''{\cdot}\beta''')$ is defined and $\beta'''{:}E_i = \alpha''{:}E_i$ and $D^*(\beta{\cdot}\beta''{\cdot}\beta''') = D^*(\beta{\cdot}\beta'{\cdot}\alpha'')$. But $D^*(\beta{\cdot}\beta') = D^*(\alpha{\cdot}\alpha')$. So for every sequence $\alpha{\cdot}\alpha'$ of type $(\varepsilon^*H\varepsilon^*)^*$ such that $|(\alpha{\cdot}\alpha'){:}E_i| = n+1$, there is a sequence $\beta{\cdot}\beta''$ of type $(H_S+(N_O{}^*N_SH_r))^*$ such that $D^*(\beta{\cdot}\beta'')$ is defined and $(\alpha{\cdot}\alpha'){:}E_i = (\beta{\cdot}\beta''){:}E_i$ and the following is true:

> for every sequence α'' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\alpha{\cdot}\alpha'{\cdot}\alpha'')$ is defined, there is a sequence β''' of type $\varepsilon^*H\varepsilon^*$ such that $D^*(\beta{\cdot}\beta''{\cdot}\beta''')$ is defined and $\alpha''{:}E_i = \beta'''{:}E_i$ and $D^*(\alpha{\cdot}\alpha'{\cdot}\alpha'') = D^*(\beta{\cdot}\beta''{\cdot}\beta''')$

QED.

Algorithm 4 generates a PEG by traversing sequences of type $\varepsilon^*H\varepsilon^*$ from each node. By Theorem 3.7, it can be modified to traverse sequences of type $H_S+(N_O{}^*N_SH_r)$ only. This yields Algorithm 5:

```
genPEG(X, B, i)
1.    R := (σ10, ...., σn0, λ, ...., λ);/* initial global state */
2.    W := {R};              /* working set of global states to analyze */
3.    A := ∅;                /* set of global states already analyzed */
4.    while (W ≠ ∅)
5.    {     V := a member of W;
6.          W := W - V;
7.          A := A ∪ {V};
8.          if (error_state(V, B))
9.          {     error(V) := TRUE;
            }
10.         else
11.         {     error(V) := FALSE;
12.               for every error-free or error-terminated, cycle-free executable
                  sequence α of type HS+NO*NSHF that begins at V
13.               {     V' := D*(V, α);
14.                     if there is no edge labelled e from V to V',
                        where e is label of the H event;
15.                     {     create edge labelled e from V to V';
                        }
16.                     if ((V' ∉ A) and (V' ∉ W))
17.                     {     W := W ∪ {V'};
                        }
                  }
            }
      }
```

Algorithm 5: More Efficient PEG Generation During Reachability Analysis

As stated earlier, a PEG generated by Algorithm 5 will be called a PEG_5, where the distinction is required. References to the global state mapped to a PEG_5 node refer to the obvious mapping arising from the fact that, for every V added to A at statement 7 of Algorithm 5, V is both a global state and a PEG_5 node.

The advantage of Algorithm 5 compared to Algorithm 4 is that Algorithm 5 traverses fewer sequences from each node. As an example, consider Protocol 1 again (Figure 3.1). Figure 3.8 indicates the type of each edge in the RG of Protocol 1:

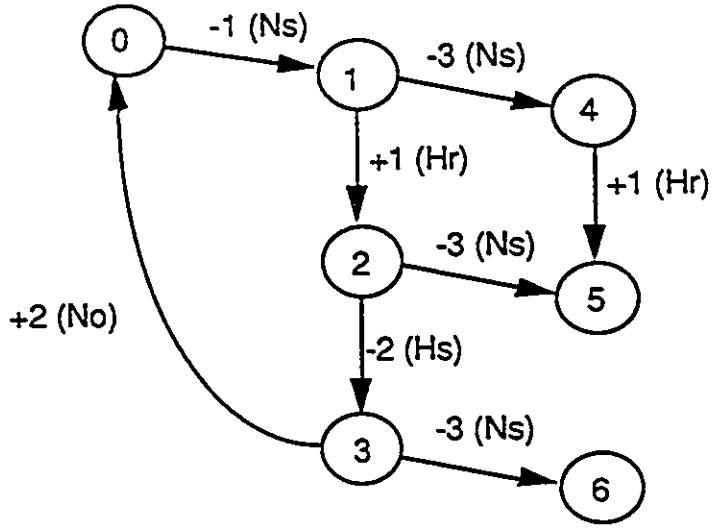

Figure 3.8 - Reachability Graph for Protocol 1 with Event Types Shown

Algorithm 5, applied to Protocol 1, proceeds as follows:

1) Set A empty and store node 0 (i.e. ℝ) in W.

2) Remove node 0 from W and add it to A. From node 0 there is no H_S event, but there is a sequence of type $N_O{}^*N_SH_r$ (-1,+1) to node 2. So node 2 is added to W, and a +1 edge is drawn from node 0 to node 2.

3) Remove node 2 from W and add it to A. From node 2 there is an H_S event (-2), leading to node 3, but no sequence of type $N_O{}^*N_SH_r$. So node 3 is added to W, and a -2 edge is drawn from node 2 to node 3.

4) Remove node 3 from W and add it to A. From node 3 there is no H_S event, but there is a sequence of type $N_O{}^*N_SH_r$, (+2,-1,+1) to node 2, so a +1 edge is drawn from node 3 to node 2. Node 2 is already in A, so it is not added to W. Now W is empty, and the algorithm terminates.

The resultant PEG is shown in Figure 3.9:

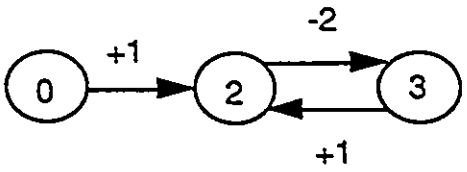

Figure 3.9 - PEG Generated by Applying Algorithm 5 to Protocol 1

Table 3.3 compares Algorithms 4 and 5, applied to Protocol 1:

	States Visited	Sequences Traversed	Edges Traversed	PEG Nodes	PEG Edges
Algorithm 4	7	15	37	7	12
Algorithm 5	3	3	6	3	3

Table 3.3 - Comparison of Algorithms 4 and 5 Applied to Protocol 1

Algorithm 5 has three main advantages over Algorithm 4:

1) fewer global states are visited

2) fewer sequences are traversed from each global state visited

3) the generated PEG is smaller

Algorithm 5 does not specify how sequences of type $H_S+N_O*N_SH_r$ are to be detected from a given node. Sequences of type H_S can be detected by the presence of a specified host send transition. If there is no specified host receive from the current state, then there can be no sequence of type $N_O*N_SH_r$. If there is a specified host receive, then sequences of type $N_O*N_SH_r$ could be detected by a restricted form of reachability analysis from the current node. In the two-process case, this would consist of running the non-host process through its input queue to identify corresponding sequences of type N_r*N_S.

4. APPLICATIONS OF PROCESS EVENT GRAPHS

4.1 Validating Process Effectiveness

As noted in the introduction, a process is said to be *effective* if every specified event sequence of the process is executable [Okum 87]. One way to determine whether all specified sequences are executable is to determine whether the PEG and the specification graph accept the same language. Definition 2.1, the initial definition of a CFSM process, defined the transition function, δ_i, as a partial function from $\sigma_i \times E$ to σ_i. Therefore a process corresponds to a deterministic FSM, i.e. given event e and state σ, there can be no more than one e edge leaving σ. This distinction is important, since language equivalence can be determined in polynomial time for deterministic FSMs, but is PSPACE-complete for non-deterministic FSMs [Hunt 76, Gare 79]. We therefore consider the language equivalence problem for deterministic PEGs only.

Theorem 4.1: For any protocol consisting of exactly two processes, in which no state of the non-host process has both an outgoing send edge and an outgoing receive edge, the PEG_5 is deterministic.

Proof: First, consider sequences of type H_S. Since the specification graphs are deterministic, given global state V and event e of type H_S, the node $V' = D(V,e)$ is determined. Now consider sequences of type $N_O{}^*N_SH_r$. The PEG_5 is deterministic unless, given some node V, there are two distinct sequences α and α' of type $N_O{}^*N_SH_r$ such that $D^*(V,\alpha) \neq D^*(V,\alpha')$, where the H_r event in α is identical to the H_r event in α'. But in the two-process case N_O is a receive event of the non-host process, and the sequence of permissible receive events is determined by the input queue at V. So for α and α' to be different, there are two possibilities

1) the length of the $N_O{}^*$ prefix is different in α and α'

2) the N_S event is different in α and α'

If these two conditions are false, then $\alpha = \alpha'$, so $D^*(V,\alpha) = D^*(V,\alpha')$. In case (1), one $N_O{}^*$ prefix is shorter than the other. But then the sequence with the shorter $N_O{}^*$ prefix executes a non-host send when it could execute a non-host receive. This contradicts the assumption that no state of the non-host process has both an outgoing send edge and an outgoing receive edge.

Recall from Theorem 3.6 that the host input queue is empty at the beginning of every sequence traversed by Algorithm 5. Therefore, in case (2), the only message on the host input queue is the message queued by the N_S event. But if the N_S events are different, the H_r events are different too, which contradicts the assumption.

QED.

It is important to note that the existence of a state in the non-host process with both an outgoing send edge and an outgoing receive edge does not necessarily imply that the PEG_5 will be non-deterministic. As an example, consider Protocol 7 (Figure 5.1) and the PEG_5 for P_1 in Protocol 7 (Figure 5.4). State 0 of P_2 has an outgoing send edge and an outgoing receive edge, but the PEG_5 is deterministic.

To see that a non-deterministic PEG_5 may arise from a two-process protocol in which the non-host process has a state with both an outgoing send edge and an outgoing receive edge, consider Protocol 3:

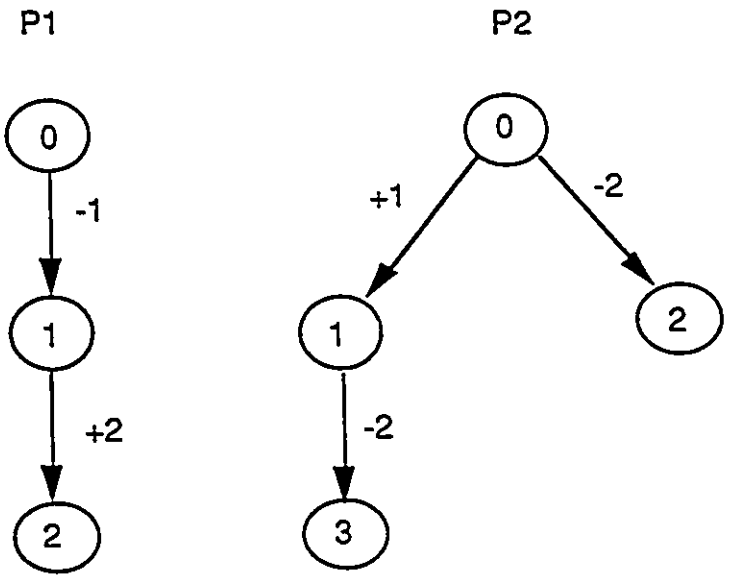

Figure 4.1 - Protocol 3

The RG for Protocol 3, and PEG_5 for Protocol 3 with P_1 as host, are shown in Figure 4.2:

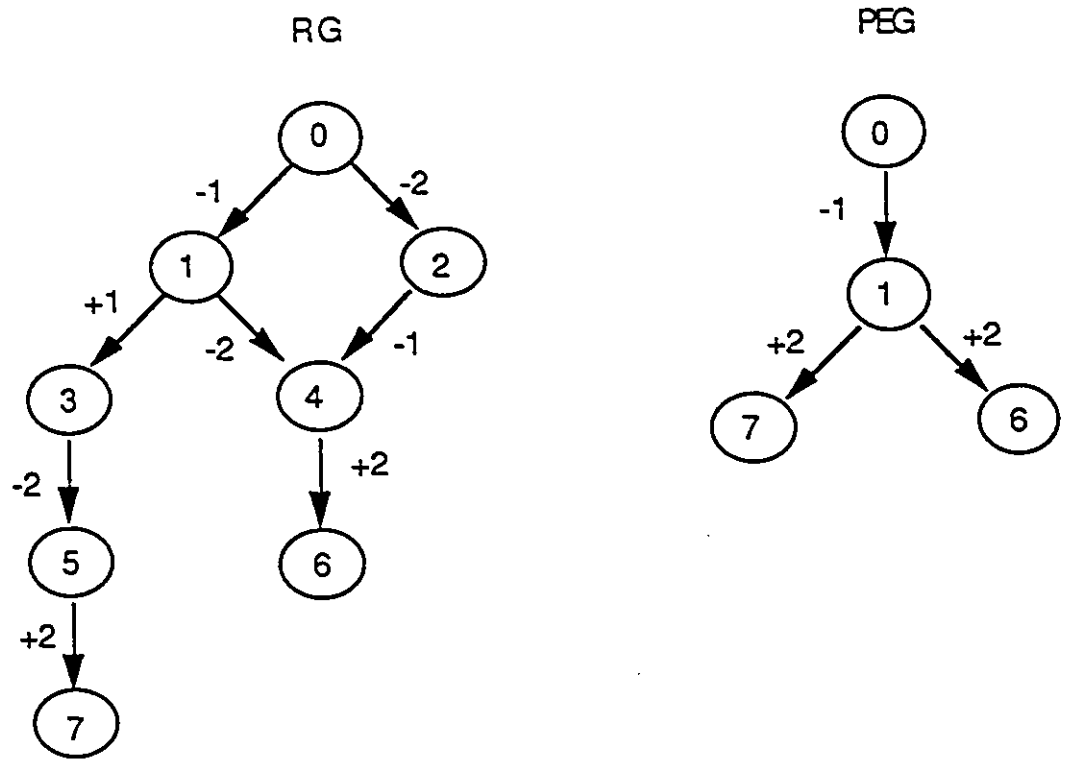

Figure 4.2 - RG for Protocol 3, and PEG$_5$ for Protocol 3 with P_1 as Host

In protocols with more than two processes, a PEG$_5$ may be non-deterministic even if no non-host process has a state with both an outgoing send edge and an outgoing receive edge. As an example, consider Protocol 4:

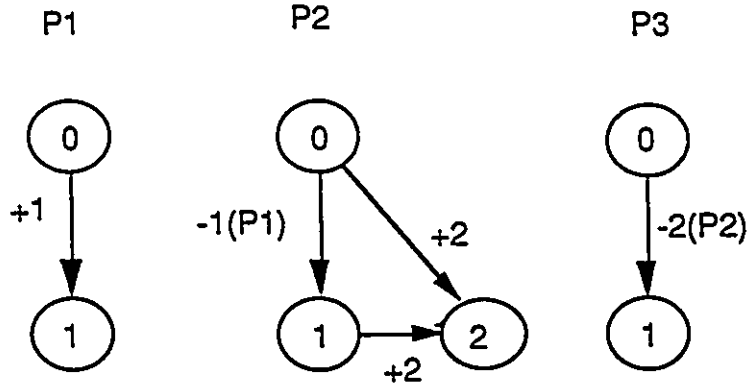

Figure 4.3 - Protocol 4

The -1(P_1) notation indicates that message 1 is being sent to P_1. The RG for Protocol 4, and PEG$_5$ for Protocol 4 with P_1 as host, are shown in Figure 4.4:

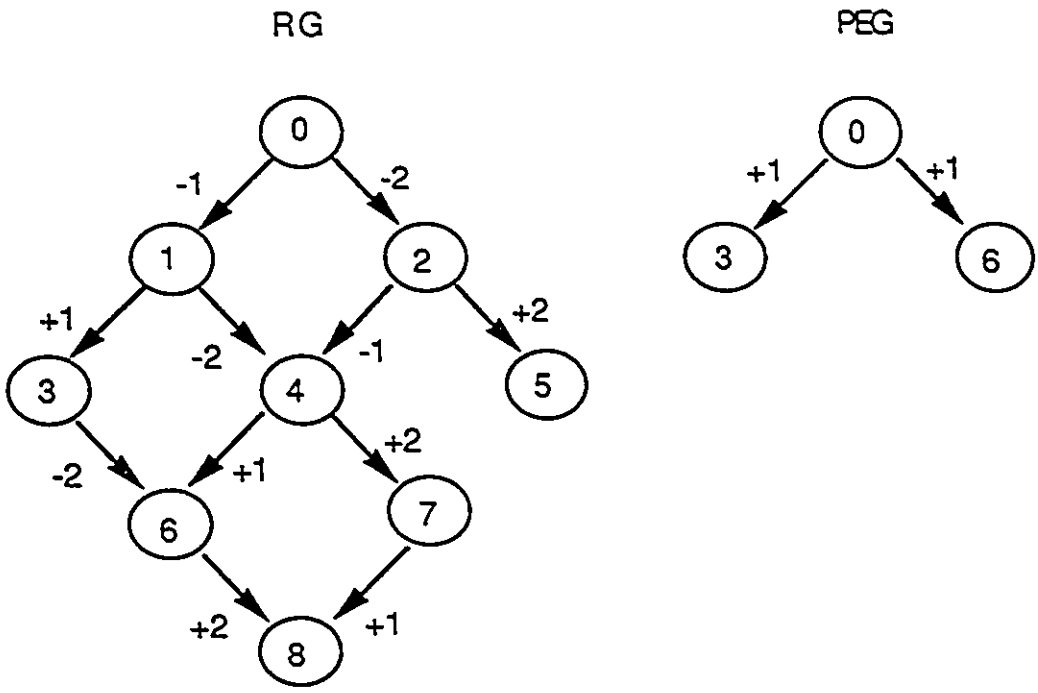

Figure 4.4 - RG for Protocol 4, and PEG5 for Protocol 4 with P_1 as Host

A trivial example of a deterministic PEG_5 in a three-process protocol can be seen by considering the PEG_5 for P_2 in Protocol 4. It is identical to the specification graph for P_2 in Protocol 4.

For deterministic PEGs, it is possible to determine in polynomial time whether the PEG is effective, i.e. whether the PEG and specification graph accept the same language. Hopcroft et al state that, for a given language L, the minimum deterministic FSM accepting L is unique up to an isomorphism [Hopc 79]. Therefore, we can determine if the PEG and specification graph accept the same language by minimizing the two graphs and checking for an isomorphism between the two minimal graphs. If there was no isomorphism, we would know that there are specified sequences that are not executable. We could begin by minimizing the PEG. If the minimal PEG had more nodes or edges than the specification graph, we would know that there are unexecutable specified sequences, and omit the specification minimization step.

For non-deterministic PEGs that are small enough, or in which the non-determinism is limited enough, we can create an equivalent deterministic PEG [Hopc 79]. Each state of the

deterministic PEG corresponds to a set of states of the non-deterministic PEG. If this could be done, then the procedures described for deterministic PEGs could be applied.

4.2 Parallel Reachability Analysis

In this section, we develop a method that introduces parallelism to reachability analysis. The method consists of generating a PEG_5, and conducting a separate partial reachability analysis for each PEG_5 node. Each partial reachability analysis explores non-host events only, and uses the global state mapped to the PEG_5 node as the initial state. For maximum parallelism, the partial reachability analysis may be started when the PEG_5 node is generated, rather than waiting until the PEG_5 is completed.

Theorem 4.2: For every $V \in R(X)$, there is a sequence α such that $D^*(\alpha) = V$, and either α is of type ε^*, or α is of type $(\varepsilon^*H\varepsilon^*)^*$.

Proof: For every $V \in R(X)$, there is at least one sequence α such that $D^*(\alpha) = V$. Either α contains some host transitions, or it does not. If it does not, then it is of type ε^*. If it does, then it is of type $(\varepsilon^*H\varepsilon^*)^*$.

QED.

Theorem 4.3: Given a mono-receiving protocol X, every global state V reachable from $\mathbb{R}$ by a sequence α of type $(\varepsilon^*H\varepsilon^*)^*$ is reachable by an ε-sequence from some global state V′ that is reachable from $\mathbb{R}$ by a sequence β of type $(H_s+(N_o{}^*N_sH_r))^*$, if error_state($D^*(\beta)$) is false for each such β.

Proof: (by induction on $|\alpha{:}E_i|$)

Basis ($|\alpha{:}E_i| = 1$): By Theorem 3.4.

Induction Step: Suppose Theorem 4.3 is true for all α such that $|\alpha{:}E_i| = n$. Consider any sequence $\alpha{\cdot}\alpha'$ such that $|(\alpha{\cdot}\alpha'){:}E_i| = n+1$, where $|\alpha{:}E_i| = n$, and $|\alpha'{:}E_i| = 1$. Therefore α' is of type $\varepsilon^*H\varepsilon^*$. Consider the β that corresponds to α in the inductive hypothesis. By the inductive hypothesis, there is some ε-sequence χ such that $D^*(\beta{\cdot}\chi) = D^*(\alpha)$. Let $\beta' = \chi{\cdot}\alpha'$. Therefore $D^*(\beta{\cdot}\chi{\cdot}\alpha') = D^*(D^*(\beta{\cdot}\chi),\alpha') = D^*(D^*(\alpha),\alpha') = D^*(\alpha{\cdot}\alpha')$. Since χ is of type ε^* and α' is of type $\varepsilon^*H\varepsilon^*$, $\beta' = \chi{\cdot}\alpha'$ is of type $\varepsilon^*H\varepsilon^*$. Therefore there is a sequence of type $\varepsilon^*H\varepsilon^*$ from $D^*(\beta)$ to $D^*(\alpha{\cdot}\alpha')$. By Theorems 3.3 and 3.4, there is a sequence β'' of type $H_s+(N_o{}^*N_sH_r)$ from $D^*(\beta)$ to some node V, such that there is an ε-

sequence from V to $D^*(\alpha \cdot \alpha')$. Therefore $\beta \cdot \beta''$ is of type $(H_S+(N_O{}^*N_SH_r))^*$, and there is an ε-sequence from $D^*(\beta \cdot \beta'')$ to $D^*(\alpha \cdot \alpha')$.

QED.

Theorem 4.4: Given a mono-receiving protocol X, every global state $V \in R(X)$ is reachable by an ε-sequence from some global state V' that is reachable from $\mathbb{R}$ by a sequence β of type $(H_S+(N_O{}^*N_SH_r))^*$, if error_state($D^*(\beta)$) is false for each such β.

Proof: From Theorem 4.2, every reachable global state is reachable from $\mathbb{R}$ by either an ε-sequence or a sequence of type $(\varepsilon^*H\varepsilon^*)^*$. From Theorem 4.3, every node reachable by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$ is reachable by an ε-sequence from a node that is reachable by a sequence of type $(H_S+(N_O{}^*N_SH_r))^*$. But nodes that are not reachable by a sequence of type $(\varepsilon^*H\varepsilon^*)^*$ from $\mathbb{R}$ are reachable by an ε-sequence from $\mathbb{R}$.

QED.

Theorem 4.4 states that, under the specified conditions, every reachable global state is reachable by an ε-sequence from some PEG_5 node. Therefore the PEG_5 can be used to introduce a degree of parallelism into the search for error states. A localized reachability analysis, restricted to non-host transitions only (i.e. ε-sequences), could be performed beginning at each PEG_5 node. These analyses could be performed in parallel, and would be guaranteed to search all reachable global states. For maximum parallelism, the search for error states from a given PEG_5 node could be started as soon as the node was generated.

As an example, consider Protocol 4 again (Figures 4.3 and 4.4). A partial reachability analysis beginning at each PEG_5 node, and considering only non-host events, is shown in Figure 4.5:

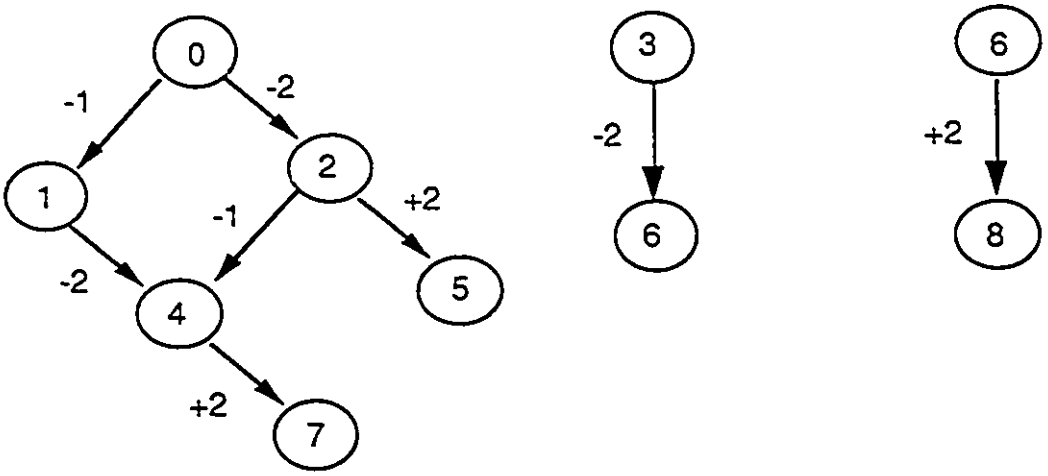

Figure 4.5 - Parallel Partial Reachability Analyses for Protocol 4

If each partial reachability analysis were started as soon as the corresponding PEG_5 node was generated, the last one to finish in the Protocol 4 example above would be the partial reachability analysis starting at PEG_5 node 0. It traverses six events, and starts after zero events have been traversed by Algorithm 5, so the parallel reachability analysis would be complete after an "elapsed time" of six events. A sequential reachability analysis would require an "elapsed time" of eleven events, as can be seen by counting the edges in the RG. Of course, a rigorous comparison of the time and space requirements of the two approaches would require a much more detailed analysis, which is outside the scope of this thesis.

It is interesting to observe that simply executing Algorithm 5 once with each process as host, i.e. generating each PEG_5, does not guarantee that every reachable global state will be visited. For example, consider Protocol 4 again. Figure 4.6 shows PEG_5s for Protocol 4 with P_2 and P_3 as hosts (the PEG_5 for P_1 was shown in Figure 4.4) :

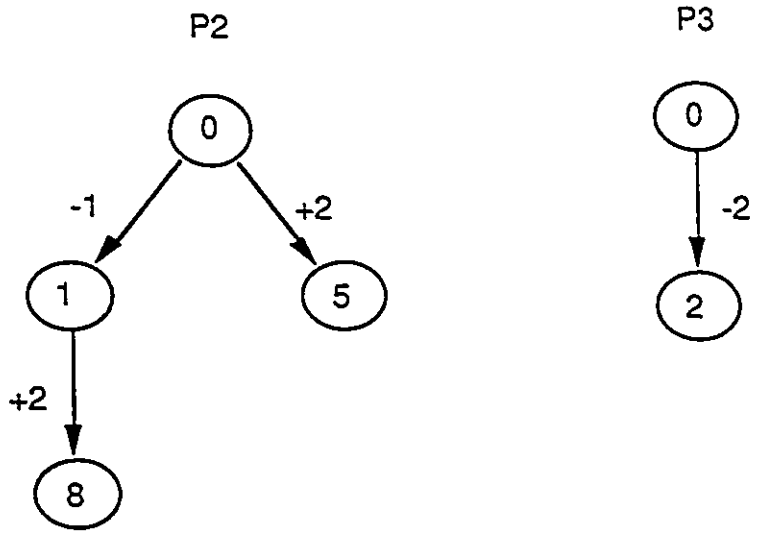

Figure 4.6 - PEGs for Protocol 4 with P2 and P3 as Hosts

Node 7 of the RG for Protocol 4 (Figure 4.4) is not visited by Algorithm 5 during any of the three PEG traversals. It is visited, however, by the parallel reachability analyses, as shown in Figure 4.5.

4.3 Detecting Process Blockage

Definition 4.1: The *maximal process language* for process P_i in protocol X given bound B, denoted $LM_i(X,B)$, is defined as follows:

$LM_i(X,B) = \{\alpha \mid \alpha \in E_i^*, \exists\, \beta \in E^*$ such that the following four conditions are true:

1) $\alpha = \beta{:}E_i$

2) β is of finite length

3) $V = D^*(\beta)$ is defined

4) there is no prefix p of β such that $|p| < |\beta|$ and error_state($D^\sim(p)$,B) = TRUE

and one of the following three conditions is true:

5) error_state($D^*(\beta)$,B) = TRUE

6) there is a non-null ε-sequence that begins at $D^*(\beta)$ and ends at an error state

7) there is an ε-cycle that begins at $D^*(\beta)$

}

Note that $L_i(X,B) \supseteq LM_i(X,B)$. Sequences in $LM_i(X,B)$ are called *maximal sequences*, and correspond to possible *process blockage* of P_i. After P_i executes some $\alpha \in LM_i(X,B)$, either an error state may be reached or an infinite sequence may be executed without further participation by P_i.

Definition 4.2: A PEG G is said to be a marked PEG (MPEG) for process P_i given bound B iff there is a node r (the "initial node") of G, and a set of marked nodes of G, such that the set of directed sequences beginning at r and ending at a marked node is identical to $LM_i(X,B)$.

Marking a PEG to generate an MPEG reveals which of the process' executable sequences may lead to process blockage, i.e. an error state, or a state from which there is an ε-cycle. Process blockages represent failure to progress, and are therefore of general interest. Protocols with only two processes cannot contain ε-cycles, since the non-host process cannot continue to send without overflowing the blocked process' input queue, and cannot continue to receive without consuming all its input. So in the two-process case, the presence of a marked node implies the presence of an error state. But in protocols with more than two processes, it is possible for a process to be blocked without the presence of an error state.

From Definition 4.1, it is clear that a PEG node V must be marked if any of three conditions hold:

1) error_state(V,B) = TRUE

2) there is a non-null ε-sequence from V to some node V′ such that error_state(V′,B) = TRUE

3) there is an ε-cycle that begins at V

Therefore an MPEG can be generated by applying the following algorithm to a PEG[5]:

```
genMPEG(PEG,B)
1.      for every node V in PEG
2.      {       if  error_state(V,B)
3.              {       marked(V) := TRUE;
                }
4.              else if there is an ε-cycle from V
5.              {       marked(V) := TRUE;
                }
6.              else if ∃ V' reachable by an ε-sequence from V
                such  that  error_state(V',B)
7.              {       marked(V) := TRUE;
                }
8.              else
9.              {       marked(V) := FALSE;
                }
        }
```

Algorithm 6: Convert Output of Algorithm 5 to an MPEG

As an example, consider the RG for Protocol 1 (Figure 3.2), with P_1 as host. Nodes 5 and 6 of the RG represent error states. Assuming B > 1, error_state is false for all other nodes. The RG contains no ε-cycles (since Protocol 1 has only two processes, no ε-cycles are possible). The PEG_5 for P_1 in Protocol 1 contains RG nodes 0, 2, and 3 (Figure 3.9). There is an ε-sequence from node 2 to node 5, and from node 3 to node 6, but not from node 0 to nodes 5 or 6. Therefore, only nodes 2 and 3 would be marked in the MPEG for Protocol 1, yielding the MPEG shown in Figure 4.10 (bold face is used to indicate a marked node):

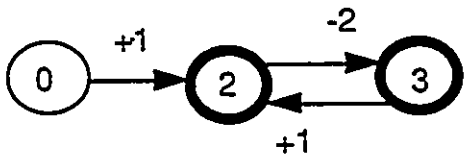

Figure 4.7 - MPEG for Process P_1 in Protocol 1

The MPEG in Figure 4.7 shows that P_1 will always be able to execute at least one +1 event, but may be blocked after any subsequent event.

Some process blockages may occur only in circumstances that are not fair, i.e. the host process has a transition it can execute, but never gets a chance because the other processes are able to execute transitions forever. In some cases, unfair sequences may not be of interest, because the intended implementation environment may be assumed to be fair. To

reflect fairness, Algorithm 6 could count ε-cycles only when there is no possible exit from the cycle that leads to a host transition.

5. RELATED ALGORITHMS

This chapter compares Algorithm 5 with two related algorithms: the maximal progress algorithm [Goud 84] and the YK-algorithm [Yuan 89]. These two algorithms are superficially similar to Algorithm 5, in that both validate a CFSM-based protocol asymmetrically, i.e. with a specified host process. The intent of this section is to point out the ways in which these algorithms differ from Algorithm 5.

5.1 The Maximal Progress Algorithm

A major problem with reachability analysis is that the number of reachable global states is often too large to explore. This is known as the "state space explosion problem" [Rubi 82, Goud 84, Lin 87, Yuan 88, Holz 91]. The maximal progress algorithm [Goud 84], is a partial solution to the state space explosion problem. It addresses the problem by traversing only part of the reachable global state space, and is therefore classified as a "partial search technique" [Lin 87]. The partial search generated by the maximal progress algorithm advances the host process as far as possible at each step. As in reachability analysis, each state generated is examined for possible state errors, i.e. deadlock, unspecified reception and queue overflow. Maximal progress is essentially a heuristic approach that is targeted at generating queue overflows. This is contrasted with the fair progress algorithm [Rubi 82], which is targeted at generating deadlocks, and which does not employ the concept of a host process. Unlike Algorithm 5, the maximal progress algorithm is specified only for two-process protocols, and requires every node in the process specification graphs to have at least one outgoing edge.

Using our notation, the maximal progress algorithm is specified below. The phrase "edges from v_i" refers to transitions from node v_i in the process specification graph for P_i. The phrase "add D(V,e) to W, if not found", is to be interpreted as follows:

```
V' := D(V, e);
if ((V' ∉ A) and (V' ∉ W))
{       W := W ∪ {V'};
}
```

The rm(V) variable keeps track of nodes that have been marked "receiving-mixed", as explained in [Goud 84]. Since there are assumed to be only two processes, there is only one non-host process, for which the subscript j is used.

```
analyze(X, B, i)
1.    R := (σ10, σ20, λ, λ);        /* initial global state */
2.    W := {R};                     /* working set of global states to analyze */
3.    A := ∅;                       /* set of global states already analyzed */
4.    error := FALSE;
5.    rm(R) := FALSE;               /* R is not a "receive-mixed" state */
6.    while ((W ≠ ∅) and not error)
7.    {     V := (vi,vj,qi,qj) is a member of W;
8.          W := W - V;
9.          A := A ∪ {V};
10.         if (error_state(V,B)) error := TRUE;
11.         else if (not rm(V))
12.         {     if ((every edge from vi is of type HS)
                    or ((∃ edges of type Hr from vi) and (qi ≠ λ)))
13.               {     for every executable edge e of type HS+Hr from V
14.                     {     rm(D(V,e)) := FALSE;
15.                           add D(V,e) to W, if not found;
                      }
                  }
16.               else if  ((every edge from vi is of type Hr) and (qi = λ))
17.               {     for every executable edge e of type NS+Nr from V
18.                     {     rm(D(V,e) := FALSE;
19.                           add D(V,e) to W, if not found;
                      }
                  }
20.               else if ((∃ edges of type Hr and HS from vi) and (qi = λ))
21.               {     for every executable edge e from V
22.                     {     rm(D(V,e)) := FALSE;
23.                           if (e is of type NS+Nr)
24.                           {     rm(D(V,e)) := TRUE;
                              }
25.                           add D(V,e) to W, if not found;
                      }
                  }
            }
26.         else /* error_state(V,B) = FALSE and rm(V) = TRUE */
27.         {     if (qi ≠ λ)
28.               {     for every executable edge e of type Hr from V
29.                     {     rm(D(V,e)) := FALSE;
30.                           add D(V,e) to W, if not found;
                      }
                  }
31.               else if ((∃ edges of type NS from vj) or (qj ≠ λ))
32.               {     for every executable edge e of type NS+Nr from V
33.                     {     rm(D(V,e)) := TRUE;
34.                           add D(V,e) to W, if not found;
                      }
                  }
            }
      }
```

The Maximal Progress Algorithm

To show that the maximal progress algorithm traverses the RG in a different way than Algorithm 5 does, we will use the sample protocol presented in [Goud 84]. The protocol, which we will call Protocol 5, is as follows:

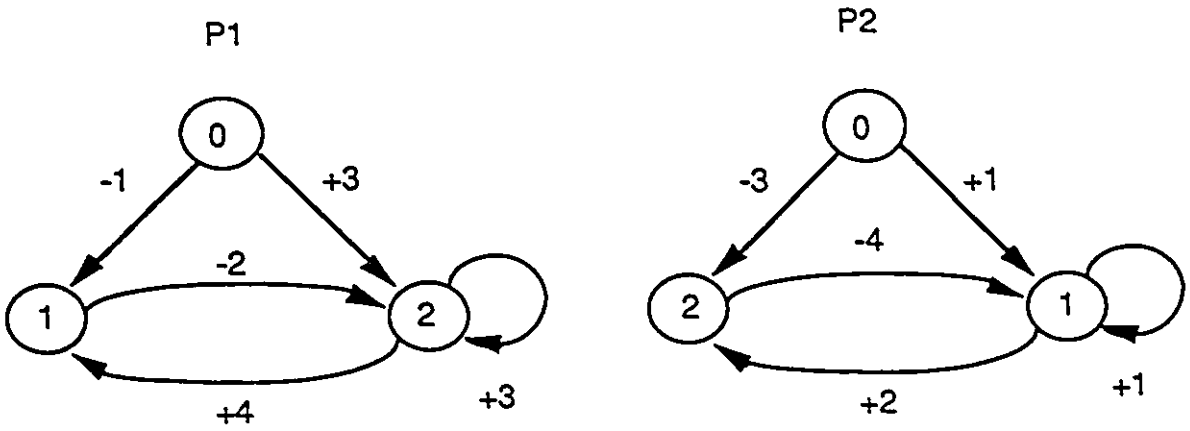

Figure 5.1 - Protocol 5

To illustrate the RG traversals, nodes will be labelled with the information defining the corresponding global state, as a vector in the form (v_1,v_2,q_1,q_2). Since all messages have single digit identifiers, the queue contents will simply be represented as a string of digits, with the leftmost digit representing the head of the queue. The traversal generated by the maximal progress algorithm with P_1 as host is shown in Figure 5.2 (an isomorphic graph is given in [Goud 84]):

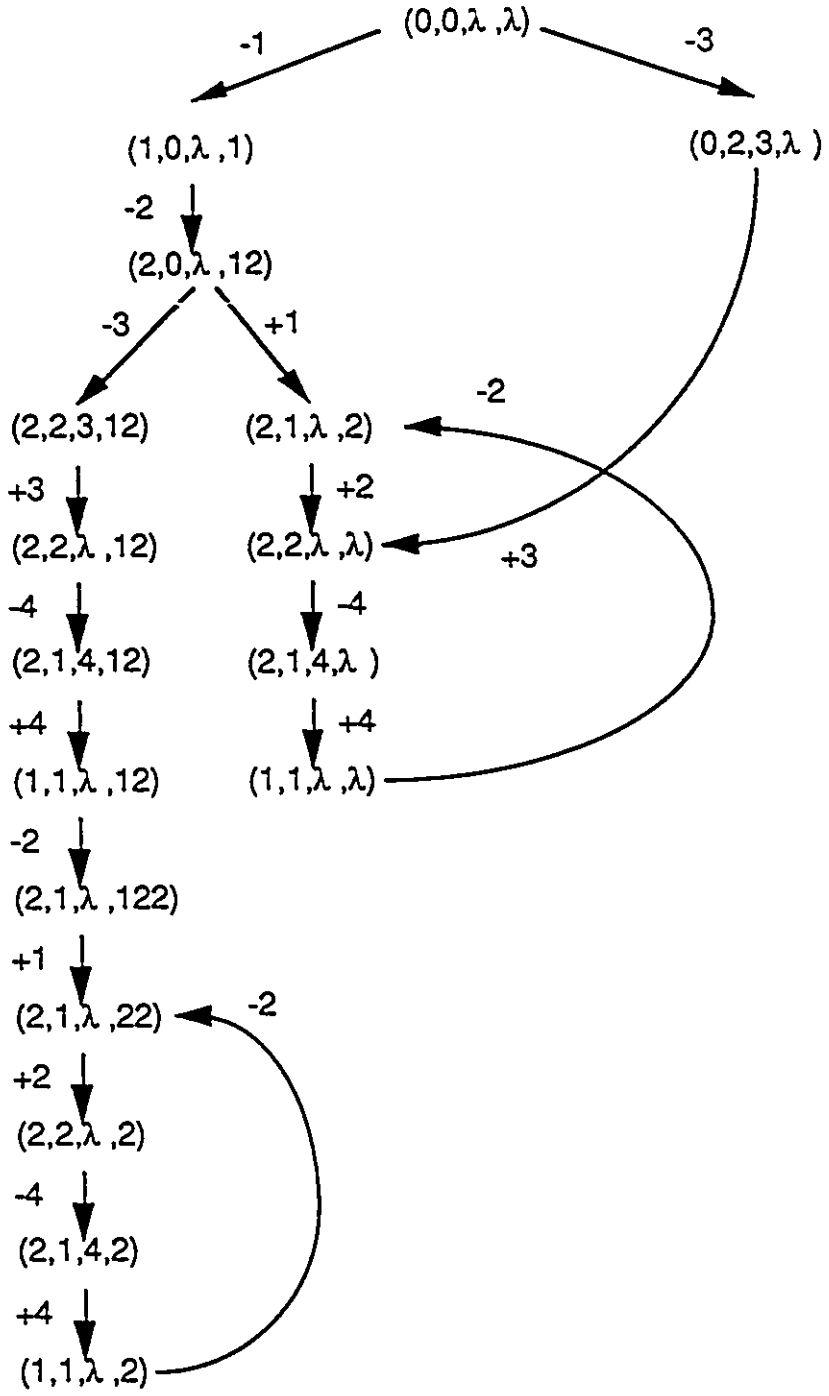

Figure 5.2 - Protocol 5 Traversal By Maximal Progress with P_1 as Host

The traversal generated by Algorithm 5 for Protocol 5, with P_1 as host, is as follows:

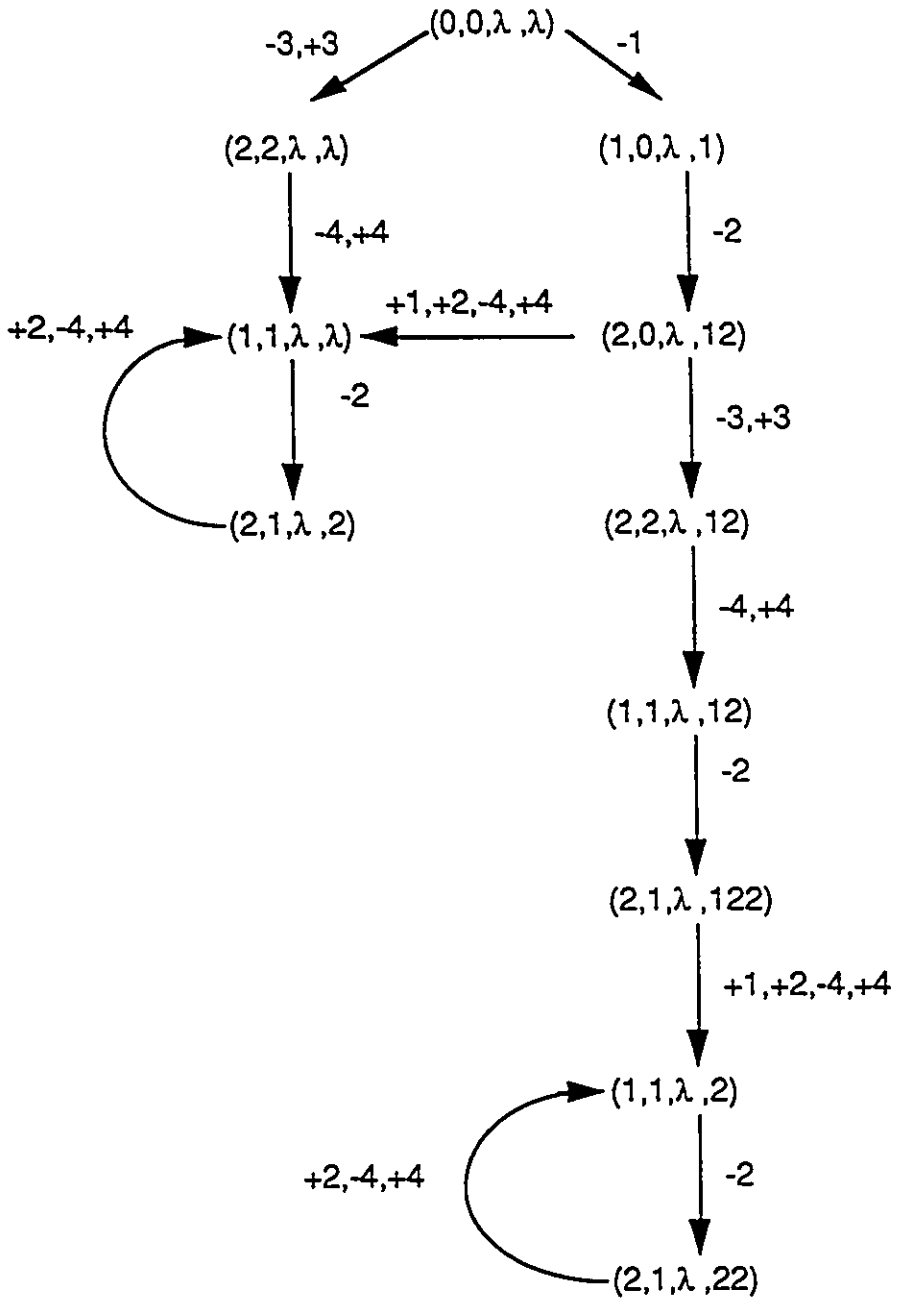

Figure 5.3 - Protocol 5 Traversal By Algorithm 5 with P1 as Host

The reachability graph for Protocol 5 is given in [Goud 84]. The following table compares the maximal progress traversal with the Algorithm 5 traversal, and includes reachability analysis numbers for comparison:

	Nodes Stored	Edges Traversed
Maximal Progress	17	19
Algorithm 5	11	27
Reachability Analysis	29	47

Table 5.1 - Comparison of Algorithm 5 with Related Algorithms

Displaying only the host transitions of Figure 5.3 and renaming the nodes yields the PEG_5, which is as follows:

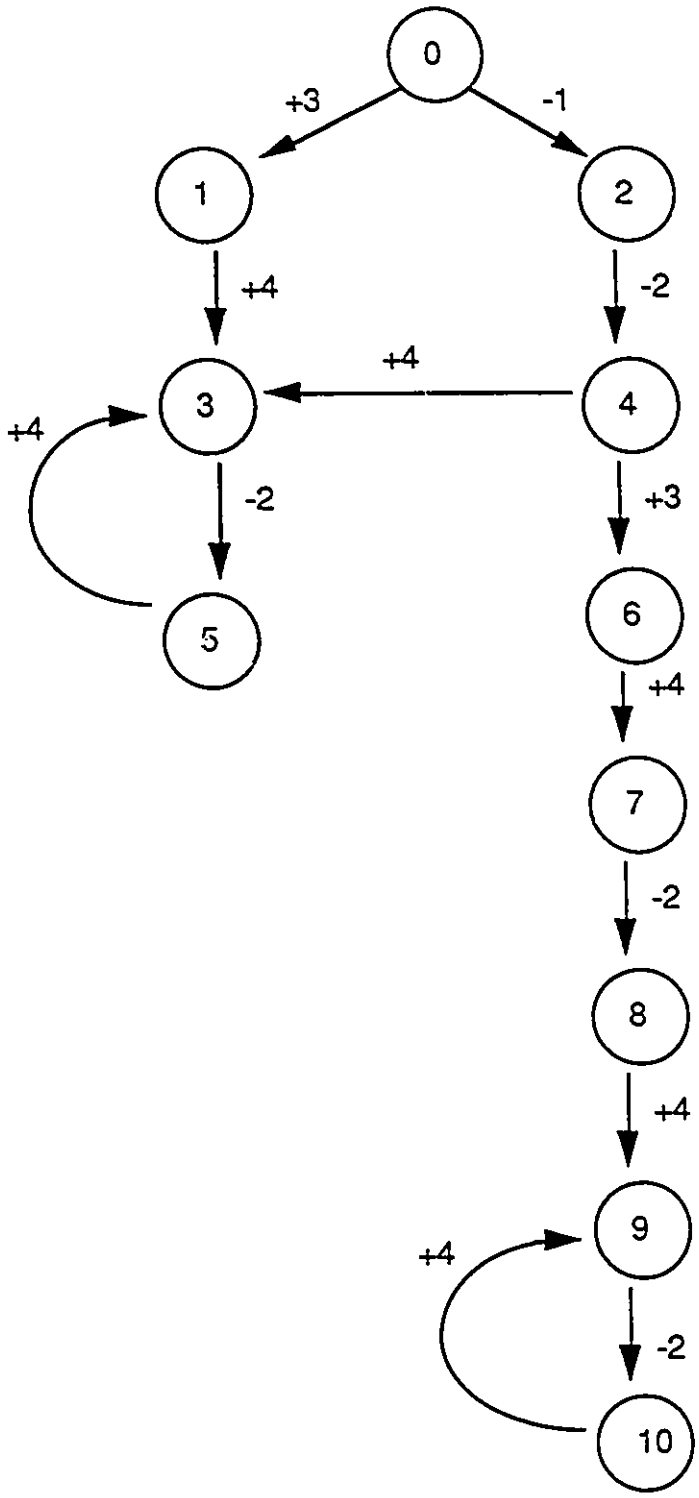

Figure 5.4 - PEG5 for P_1 in Protocol 5

For a better understanding of the behaviour of P_1 in Protocol 5, we would minimize the PEG. The FSM minimization algorithm presented in [Hopc 79] is reproduced below. The algorithm operates on a table containing an entry for every pair of distinct states. The variable *entry(p,q)* refers to the table entry for states p and q. On completion of the algorithm, entries that are marked with an X correspond to pairs of states that have been *distinguished*, according to the following definition:

Definition 5.1: A state p is said to be *distinguished* from state q iff $\exists$ a sequence α such that exactly one of $\delta^*(p,\alpha)$ and $\delta^*(q,\alpha)$ is a final state, where δ^* is the iterated transition function of the FSM.

The algorithm also associates a set of dependent state pairs with each entry. The variable *dependent(p,q)* is a set containing all the pairs of states that will be known to be distinguished if p and q are known to be distinguished. The FSM minimization algorithm is as follows:

```
minimizeFSM(G)     /* G = (V,E), i.e. (V=states,E=transitions) */
1.      for every pair p, q ∈ V, p ≠ q
2.      {       entry(p,q) := BLANK;
3.              dependent(p,q) := ∅;
4.              if exactly one of p and q is a final state
5.              {       entry(p,q) := X;
                }
        }
6.      for every pair p, q ∈ V, p ≠ q
7.      {       if (entry(p,q) = BLANK)    /* p, q not distinguished yet */
8.              {       if ∃ a such that entry(δ(p,a),δ(q,a)) = X
9.                      {       mark(p,q);
                        }
10.                     else
11.                     {       for every input symbol a
12.                             {       if (δ(p,a) ≠ δ(q,a))
13.                                     {       dependent(δ(p,a),δ(q,a)) :=
                                                dependent(δ(p,a),δ(q,a)) ∪ {(p,q)};
                                        }
                                }
                        }
                }
        }

mark(p,q)    /* called by "minimizeFSM" */
1.      entry(p,q) := X;
2.      for every pair (r,s) ∈ dependent(p,q)
3.      {       mark(r,s);
        }
```

Algorithm 7 - Finite State Machine (FSM) Minimization

Algorithm 7 makes two assumptions that do not hold for PEGs:

1) some states are not final

2) δ(p,a) is defined for every state p and input symbol (i.e. executable host event) a

Assumption (1) is used at statements 4 and 5 of Algorithm 7 to mark entries that are known to be distinguished at the beginning. But if all states are final, as in a PEG, no entries will be marked at statement 5. Therefore no entries will be marked at any step of the algorithm.

Assumption (2) is used at statements 11 and 12. Although assumption (2) is true of some PEGs, it is not true of every PEG. When it is true, every sequence of executable host events is executable. In this case, the minimal PEG consists of a single state with one edge for each executable host event, where each edge has its head and tail at the same state. When assumption (2) is false, this fact may be used to mark entries in the first phase of the algorithm.

Since every PEG node is final, the definition of distinguished states (Definition 5.1) must also be modified. We will replace it with Definition 5.2:

Definition 5.2: A state p is said to be *PEG-distinguished* from state q iff $\exists$ a sequence α such that exactly one of $\delta^*(p,\alpha)$ and $\delta^*(q,\alpha)$ is defined, where δ^* is the iterated transition function of the PEG.

From these observations we can derive Algorithm 8:

```
minimizePEG(G)      /* G = (V,E) is a PEG for Pi in X */
1.    for every pair p, q ∈ V, p ≠ q
2.    {       entry(p,q) := BLANK;
3.            dependent(p,q) := ∅;
4.            if ∃ a such that exactly one of δ(p,a) and δ(q,a) is defined
5.            {       entry(p,q) := X;
              }
      }
6.    if ∃ p, q ∈ V, p ≠ q such that entry(p,q) = X
7.    {       for every pair p, q ∈ V, p ≠ q
8.            {       if (entry(p,q) = BLANK)
9.                    {       if ∃ a such that δ(p,a) and δ(q,a) are defined
                              and δ(p,a) ≠ δ(q,a) and entry(δ(p,a),δ(q,a)) = X
10.                           {       mark(p,q);
                              }
11.                           else
12.                           {       for every a such that δ(p,a) and δ(q,a)
                                      are defined and δ(p,a) ≠ δ(q,a)
13.                                   {       dependent(δ(p,a),δ(q,a)) :=
                                              dependent(δ(p,a),δ(q,a)) ∪ {(p,q)};
                                      }
                              }
                      }
              }
      }
14.   else
15.   {       create single-state minimal PEG, as described above;
      }

mark(p,q)     /* called by "minimizePEG" */
1.    entry(p,q) := X;
2.    for every pair (r,s) ∈ dependent(p,q)
3.    {       mark(r,s);
      }
```

Algorithm 8 - PEG Minimization

Now we will apply Algorithm 8 to the PEG_5 for P_1 in Protocol 7 (Figure 5.4). The table, after marking entries in the first phase (statements 1 to 5 of Algorithm 8) is shown in Table 5.2:

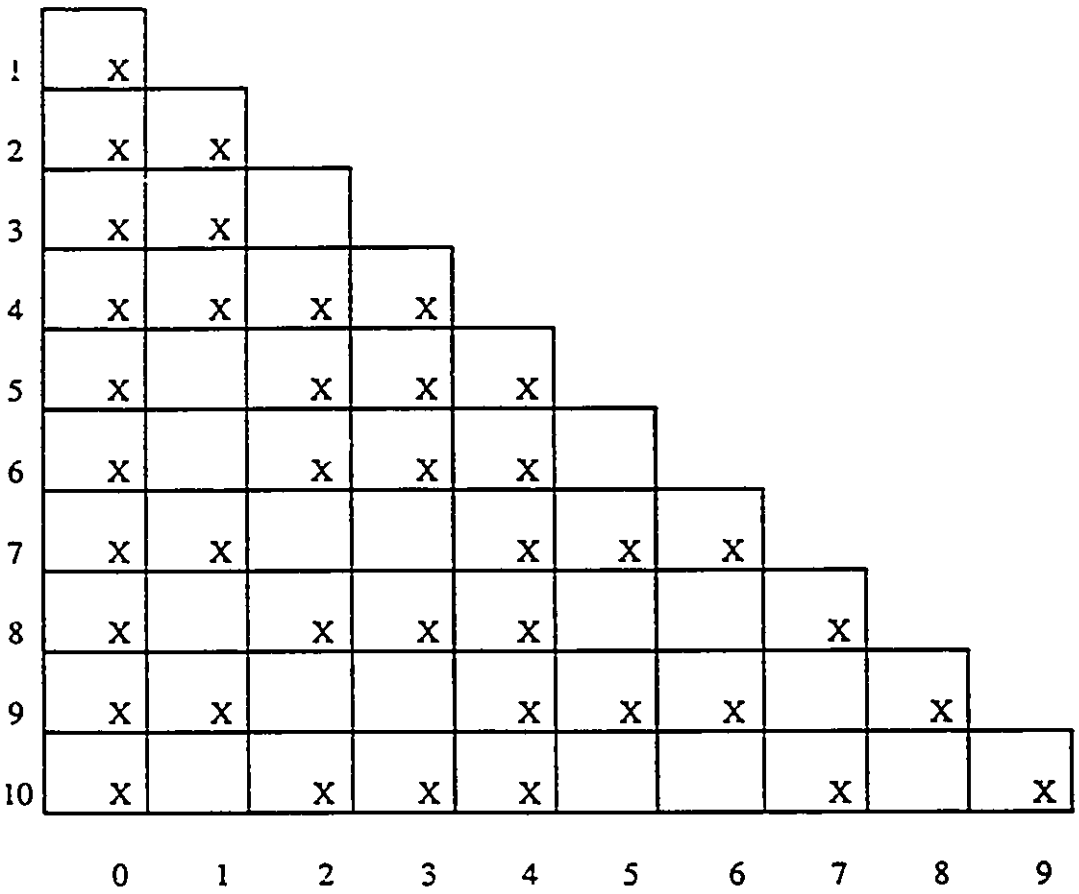

Table 5.2 - Initialized Table in Minimization of PEG_5 for P_1 in Protocol 5

The second phase of Algorithm 8 (statements 6 to 15) examines the sixteen blank entries: (1,5), (1,6), (1,8), (1,10), (2,3), (2,7), (2,9), (3,7), (3,9), (5,6), (5,8), (5,10), (6.8), (6,10), (7,9), and (8,10). This proceeds as follows:

1) (1,5): $\delta(1,+4) = \delta(5,+4)$, so no action is taken.

2) (1,6): $\delta(1,+4) = 3$, $\delta(6,+4) = 7$. entry(3,7) is blank, so set dependent(3,7) = {(1,6)}.

3) (1,8): $\delta(1,+4) = 3$, $\delta(8,+4) = 9$. entry(3,9) is blank, so set dependent(3,9) = {(1,8)}.

4) (1,10): $\delta(1,+4) = 3$, $\delta(10,+4) = 9$. entry(3,9) is blank, so set dependent(3,9) = {(1,8),(1,10)}.

5) (2,3): $\delta(2,-2) = 4$, $\delta(3,-2) = 5$. entry(4,5) is marked, so mark(2,3). dependent(2,3) = $\emptyset$, so just set entry(2,3) = X.

6) (2,7): $\delta(2,-2) = 4$, $\delta(7,-2) = 8$. entry(4,8) is marked, so mark(2,7). dependent(2,7) = $\emptyset$, so just set entry(2,7) = X.

7) (2,9): $\delta(2,-2) = 4$, $\delta(9,-2) = 10$. entry(4,10) is marked, so mark(2,9). dependent(2,9) = $\emptyset$, so just set entry(2,9) = X.

8) (3,7): $\delta(3,-2) = 5$, $\delta(7,-2) = 8$. entry(5,8) is blank, so set dependent(5,8) = {(3,7)}.

9) (3,9): $\delta(3,-2) = 5$, $\delta(9,-2) = 10$. entry(5,10) is blank, so set dependent(5,10) = {(3,9)}.

10) (5,6): $\delta(5,+4) = 3$, $\delta(6,+4) = 7$. entry(3,7) is blank, so set dependent(3,7) = {(1,6),(5,6)}.

11) (5,8): $\delta(5,+4) = 3$, $\delta(8,+4) = 9$. entry(3,9) is blank, so set dependent(3,9) = {(1,8),(1,10),(5,8)}.

12) (5,10): $\delta(5,+4) = 3$, $\delta(10,+4) = 9$. entry(3,9) is blank, so set dependent(3,9) = {(1,8),(1,10),(5,8),(5,10)}.

13) (6,8): $\delta(6,+4) = 7$, $\delta(8,+4) = 9$. entry(7,9) is blank, so set dependent(7,9) = {(6,8)}.

14) (6,10): $\delta(6,+4) = 7$, $\delta(10,+4) = 9$. entry(7,9) is blank, so set dependent(7,9) = {(6,8),(6,10)}.

15) (7,9): $\delta(7,-2) = 8$, $\delta(9,-2) = 10$. entry(8,10) is blank, so set dependent(8,10) = {(7,9)}.

16) (8,10): $\delta(8,+4) = \delta(10,+4)$, so no action is taken.

Steps 1 to 16 above marked the following entries: (2,3), (2,7), and (2,9). This leaves thirteen blank entries: (1,5), (1,6), (1,8), (1,10), (3,7), (3,9), (5,6), (5,8), (5,10), (6,8), (6,10), (7,9), and (8,10). A blank entry indicates that the corresponding pair of states cannot be PEG-distinguished. In other words, nodes 1, 5, 6, 8, and 10 can be merged into

a single node, and nodes 3, 7, and 9 can be merged into a single node. Identifying the merged nodes as nodes 1 and 3 respectively yields the following minimized PEG:

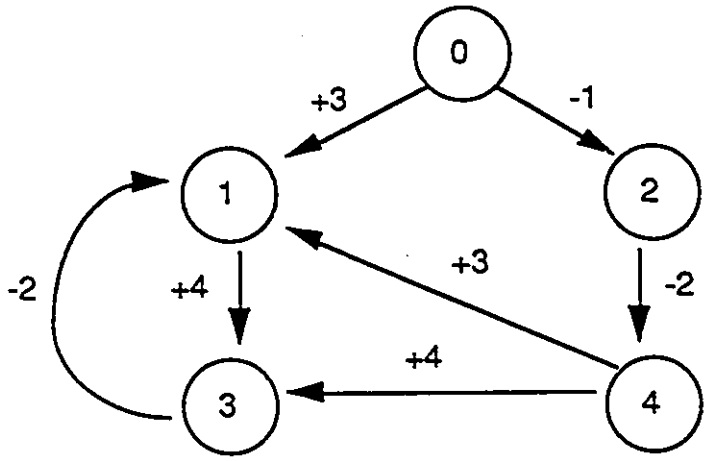

Figure 5.5 - Minimized PEG for P_1 in Protocol 5

Since the minimal PEG has more nodes than the specification graph for P_1 (Figure 5.1), they are not isomorphic, so there must be unexecutable specified sequences. Beginning a systematic search, we discover an unexecutable specified sequence of length 2: (+3,+3). On a more general level, the minimal PEG reveals that P_1 will eventually enter a (+4,-2) loop with no exit, and there are three sequences that lead to it: (+3), (-1,-2), and (--1,-2,+3). That is, L_1(Protocol 5,B) = ((+3)+(-1,-2)+(-1,-2,+3))(+4,-2)*, assuming B > 2.

5.2 The YK-Algorithm

The YK-algorithm [Yuan 89], like the maximal progress algorithm, is a variation of reachability analysis that employs the concept of a host process. The errors that it detects are deadlock, queue overflow, unspecified reception, and unexecutable transition. To detect these errors, the YK-algorithm must be executed once with each process as host. The YK-algorithm is derived from an algorithm by Kakuda et al [Kaku 86], which was derived from [Bran 83]. We will present a simplified formulation of the YK-algorithm, retaining its essential features but omitting the final phase that checks for unspecified receptions and unexecutable transitions, and compare its behaviour with the behaviour of Algorithm 5 on a

sample protocol. The YK-algorithm was presented for the two-process case only, and we will make no attempt to generalize it to cover other cases. The formulation of the algorithm given below is similar to the formulation given in [Huus 91].

The main terms used in the description of the YK-algorithm are as follows:

least state: a state in the non-host process, representing minimal progress of the non-host process. When the host process executes a send, the least state does not change. For the host to execute a receive given an empty input queue, a least state is generated for each executable sequence in the non-host process that ends with a send of the received message, and contains no other sends.

least state pair: a pair consisting of the least state, and the contents of the non-host process' input queue at the least state.

synched state: a non-host process state that can be reached from a least state pair by following an ε-sequence that ends by emptying the non-host process' input queue.

synched state pair: a pair consisting of the synched state, and the contents of the host process' input queue at that state.

synched set: the set of all synched state pairs that can be reached from a given least state pair.

The main variables used in the description of the YK-algorithm are as follows:

DL(V): flag marking potential deadlock at node V.

terminal(V): flag marking node V as a leaf node.

s: host process' state.

L = (*l*,CO): least state pair. *l* is the least state, and CO represents the contents of the non-host process' input queue.

Y = {(y_k, CI_k)}: synched set, where k ranges from 1 to the queue bound, B. y_k is the synched state and CI_k represents the contents of the host process' input queue.

V = (s, L, Y): 3-tuple associated with a node stored by the algorithm. The (s,L,Y) format represents a set of global states. The host is in state s with its input queue empty, and the non-host has at least reached L, perhaps with messages on its input queue. From L, the non-host can reach any of the states in Y by following a sequence that empties its input queue, and generates the output indicated in Y.

i, j: P_i is the host process, P_j is the non-host process.

The pseudo-code for the YK-algorithm is as follows:

```
analyze(X, B, i)
1.    R := (0,(0,λ),∅);   /* initial node */
2.    W := {R};           /* working set of global states to analyze */
3.    A := ∅;             /* set of nodes already examined */
4.    while (W ≠ ∅)
5.    {     V := (s,L,Y) is a member of W;
6.          W := W - V;
7.          checkTerminal(V);
8.          if (not terminal(V)) createChildren(V);
9.          A := A ∪ {V};
      }
10.   for every V ∈ A     /* V = (s,L=(l,CO),Y=((yk,Clk)) */
11.   {     if DL(V)
12.         {     report deadlock detected;
            }
13.         if length(CO) > B
14.         {     report queue overflow detected;
            }
      }

createChildren(V)                 /* V = (s,L=(l,CO),Y=((yk,Clk))) */
1.    for every edge e from s     /* all specified host edges */
2.    {     if e is of type Hs    /* e is a specified host send */
3.          {     createSendChild(V,e);
            }
4.          else                  /* e is a specified host receive */
5.          {     createReceiveChildren(V,e);
            }
      }
6.    if ∃ at least one child V' of V such that not DL(V')
7.    {     for every child V' of V
8.          {     DL(V') := FALSE;
            }
      }

createSendChild(V, e)       /* V = (s,L=(l,CO),Y=((yk,Clk))) */
1.    s' := δi(s,e);
2.    l' := l;
3.    CO' := CO·(message sent by e);
4.    create V' = (s',(l',CO'),∅) & attach V' as a child of V;
5.    createSynchedSet(V');
6.    DL(V') := FALSE;
7.    W := W ∪ {V'};
```

```
createSynchedSet(V)          /* V = (s,L=(l,CO),Y=((yk,Clk))) */
1.    Y := ∅;
2.    for every sequence α of type (Nr+Ns)*Nr from l, such that α consumes
      all messages of CO in order, and α consumes no other messages
3.    {     s' := δj*(l,α);
4.          Cl' := messages transmitted by α;
5.          add (s',Cl') to Y;
      }

createReceiveChildren(V,e) /* V = (s,L=(l,CO),Y=((yk,Clk))) */
1.    if ∃ at least one sequence α of type Nr*Ns from l,
      such that the message sent in the Ns event corresponds to the message
      received in e, and α is executable given CO
2.    {     for every such sequence α
3.          {     s' := δi(s,e);
4.                l' := δj*(l,α);
5.                CO' := CO minus the messages received by α;
6.                create V' = (s',(l',CO'),∅) & attach as a child of V;
7.                if CO' ≠ λ
8.                {     createSynchedSet(V');
                  }
9.                DL(V') := FALSE;
10.               W := W ∪ {V'};
            }
      }
11.   else  /* no such α */
12.   {     s' := δi(s,e);
13.         create V' = (s',UNKNOWN,UNKNOWN) & attach V' as a child of V;
14.         DL(V') := TRUE;
      }

checkTerminal(V)          /* V = (s,L=(l,CO),Y=((yk,Clk))) */
1.    if ∃ V'=(s',L'=(l',CO'),Y'=((y'k,Cl'k))) ∈ A such that
      ((s' = s) and ((L' = L) or ((Y ≠ ∅) and (Y' ⊇ Y))))
2.    {     terminal(V) := TRUE;
      }
3.    else if (DL(V)
4.    {     terminal(V) := TRUE;
      }
5.    else
6.    {     terminal(V) := FALSE;
      }
```

The YK-Algorithm

To illustrate the operation of the YK-algorithm, we will begin with Protocol 5 (Figure 5.1), using P_1 as host, as we did when comparing the maximal progress algorithm with Algorithm 5. A step-by-step execution of the YK-algorithm for this example is as follows:

1) To initialize, set $\mathbb{R} = (0,(0,\lambda),\varnothing)$, $W = \{\mathbb{R}\}$, and $A = \varnothing$.

2) Remove $\mathbb{R}$ from W and add it to A. CheckTerminal(V) finds no nodes in A that match V', so execute createChildren($\mathbb{R}$). From state 0, P_1 has two edges: (-1), (+3).

   a) Select (-1), and execute createSendChild($\mathbb{R}$,-1):

      i) $s' = \delta_1(0,-1) = 1$.

      ii) $l' = l = 0$.

      iii) $CO' = CO \cdot 1 = 1$.

      iv) Create $V' = (s',(l',CO'),\varnothing) = (1,(0,1),\varnothing)$ & attach V' as a child of $\mathbb{R}$. There are two $(N_r+N_s)*N_r$ sequences that receive all messages in CO': (+1), (-3,-4,+1). Select (+1) first:

         1) $s'' = \delta_2*(l,\alpha) = 1$.

         2) CI'' = messages transmitted by $\alpha = \lambda$.

         3) add $(1,\lambda)$ to Y'.

         Now select (-3,-4,+1):

         1) $s'' = \delta_2*(l,\alpha) = 1$.

         2) CI'' = messages transmitted by $\alpha = (3 \cdot 4)$.

         3) add $(1,3 \cdot 4)$ to Y'.

      v) Set $V' = (s',(l',CO'),Y') = (1,(0,1),((1,\lambda),(1,3 \cdot 4)))$.

   b) Select +3. Execute createReceiveChildren($\mathbb{R}$,+3):

      i) There is one executable sequence of type N_r*N_s ending in -3, i.e. (- -3). So select it:

         1) $s' = \delta_1(0,+3) = 2$.

2) $l' = \delta_2^*(l,(-3)) = 2$.

3) CO′ = CO minus messages received by (-3) = λ.

4) Since CO′ = λ, Y′ = ∅.

5) create V′ = (s′,(l′,CO′),Y′) = (2,(2,λ),∅), and attach V′ as a child of ℝ.

3) There is at least one child V′ of ℝ such that not DL(V′), so set DL(V′) = FALSE for all children of ℝ.

*** Note: W = {(1,(0,1),((1,λ),(1,3·4))), (2,(2,λ),∅)}

A = {(0,(0,λ),∅)}

4) Remove V = (1,(0,1),((1,λ),(1,3·4))) from W & add it to A. From state 1, P_1 has one specified edge: (-2). Execute createSendChild(V,-2).

a) $s' = \delta_1(1,-2) = 2$.

b) $l' = l = 0$.

c) CO′ = CO·2 = 1·2.

d) Y′ = ∅;

e) there are two sequences of type $(N_r+N_s)^*N_r$ from state 0 in P_2 that receive 1·2: (+1,+2), (-3,-4,+1,+2).

1) Select (+1,+2), & add (2,λ) to Y′.

2) Select (-3,-4,+1,+2), & add (2,3·4) to Y′.

f) create V′ = (s′,(l′,CO′),Y′) = (2,(0,1·2),((2,λ),(2,3·4))), attach V′ as a child of V, and add it to W.

5) There is at least one child V′ of V such that not DL(V′), so set DL(V′)=FALSE for all children of V (there is only one).

*** Note: W = {(2,(2,λ),∅), (2,(0,1·2),((2,λ),(2,3·4)))}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3·4)))}

6) Remove V = (2,(2,λ),∅) from W & add it to A. From state 2, P_1 has two specified edges: (+3),(+4).

   a) Select (+3). There are no sequences of type N_r*N_S from state 2 in P_2 that end with (-3), so create V′ = (2,UNKNOWN,UNKNOWN), mark DL(V′) = TRUE, & attach V′ as a child of V (but do not add it to W).

   b) Select (+4). There is one executable sequence of type N_r*N_S from state 2 in P_2 that ends with (-4): (-4). So create V′ = (1,(1,λ),∅), attach it as a child of V, and add it to W.

7) DL((1,(1,λ),∅)) = FALSE, so set DL((2,UNKNOWN,UNKNOWN)) = FALSE.

*** Note: W = {(2,(0,1·2),((2,λ),(2,3·4))), (1,(1,λ),∅)}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3·4))), (2,(2,λ),∅)}

8) Remove (2,(0,1·2),((2,λ),(2,3·4))) from W & add it to A. From state 2, P_1 has two specified edges: (+3),(+4).

   a) Select (+3). There is one sequence of type N_r*N_S from state 0 in P_2 that ends with (-3), i.e. (-3). So create V′ = (2,(2,1·2),∅) & attach it as a child of V. From state 2 in P_2, there is one sequence of type (N_r+N_S)*N_r that receives 1·2: (-4,+1,+2). So set V′ tc (2,(2,1·2),((2,4))) and add it to W.

   b) Select (+4). There is one executable sequence of type N_r*N_S from state 0 in P_2 that ends with (-4): (+1,+2,-4), so create V′ = (1,(1,λ),∅). But V′ = (1,(1,λ),∅) is already in W, so do not add it again (i.e. checkTerminal(V′) returns TRUE).

*** Note: W = {(1,(1,λ),∅), (2,(2,1·2),((2,4)))}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3·4))), (2,(2,λ),∅),
(2,(0,1·2),((2,λ),(2,3·4)))}

9) Remove V = (1,(1,λ),∅) from W & add it to A. From state 1, P_1 has one specified edge: (-2). So create V′ = (2,(1,2),∅). From state 1 in P_2, there is one sequence of type $(N_r+N_s)^*N_r$ that consumes 2: (+2). So V′ = (2,(1,2),((2,λ))). Attach V′ as a child of V and add it to W.

*** Note: W = {(2,(2,1-2),((2,4))), (2,(1,2),((2,λ)))}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3-4))), (2,(2,λ),∅),
(2,(0,1-2),((2,λ),(2,3-4))), (1,(1,λ),∅)}

10) Remove V = (2,(2,1-2),((2,4))) from W & add it to A. From state 2, P_1 has two specified edges: (+3),(+4).

a) Select (+3). There are no sequences of type $N_r^*N_s$ from state 2 in P_2 that end with (-3), so create V′ = (2,UNKNOWN,UNKNOWN), mark DL(V′) = TRUE, & attach V′ as a child of V (but do not add it to W).

b) Select (+4). There is one executable sequence of type $N_r^*N_s$ from state 2 in P_2 that ends with (-4): (-4). So create V′ = (1,(1,1-2),∅). There is one sequence of type $(N_r+N_s)^*N_r$ from state 1 in P_2 that consumes 1-2: (+1,+2). So set V′ = (1,(1,1-2),((2,λ))), attach it as a child of V, and add it to W.

*** Note: W = {(2,(1,2),((2,λ))), (1,(1,1-2),((2,λ)))}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3-4))), (2,(2,λ),∅),
(2,(0,1-2),((2,λ),(2,3-4))), (1,(1,λ),∅), (2,(2,1-2),((2,4)))}

11) Remove V = (2,(1,2),((2,λ))) from W & add it to A. CreateChildren(V) sets terminal(V) to TRUE when comparing it with V′ = (2,(0,1-2),((2,λ),(2,3-4))). So there is no expansion from V.

*** Note: W = {(1,(1,1-2),((2,λ)))}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3-4))), (2,(2,λ),∅),
(2,(0,1-2),((2,λ),(2,3-4))), (1,(1,λ),∅), (2,(2,1-2),((2,4))),
(1,(1,λ),((1,λ))), (2,(1,2),((2,λ)))}

12) Remove V = (1,(1,1·2),((2,λ))) from W & add it to A. From state 1, P_1 has one specified edge: (-2). So create V′ = (2,(1,1·2·2),∅). From state 1 in P_2, there is one sequence of type $(N_r+N_s)^*N_r$ that consumes 1·2·2: (+1,+2,-4,+2). So V′ = (2,(1,1·2·2),((2,4))). Attach V′ as a child of V & add it to W.

*** Note: W = {(2,(1,1·2·2),((2,4)))}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3·4))), (2,(2,λ),∅),
(2,(0,1·2),((2,λ),(2,3·4))), (1,(1,λ),∅), (2,(2,1·2),((2,4))),
(1,(1,λ),((1,λ))), (2,(1,2),((2,λ))), (1,(1,1·2),((2,λ)))}

13) Remove V = (2,(1,1·2·2),((2,4))) from W & add it to A. CreateChildren(V) sets terminal(V) to TRUE when comparing it with V′ = (2,(2,1·2),((2,4))). So there is no expansion from V.

*** Note: W = {∅}

A = {(0,(0,λ),∅), (1,(0,1),((1,λ),(1,3·4))), (2,(2,λ),∅),
(2,(0,1·2),((2,λ),(2,3·4))), (1,(1,λ),∅), (2,(2,1·2),((2,4))),
(1,(1,λ),((1,λ))), (2,(1,2),((2,λ))), (1,(1,1·2),((2,λ))),
(2,(1,1·2·2),((2,4)))}

14) Now W = ∅, so scan through A, checking every node for errors. DL(V) = FALSE for every V in A, and the maximum queue length is three, so if B > 2, no error will be reported.

The traversal generated by the YK-algorithm closely resembles that generated by Algorithm 5, as shown in Figure 5.3. Ignoring synched sets, nodes generated by the YK-algorithm correspond to the nodes of Figure 5.3 as follows:

Algorithm 5 Node	YK-Algorithm Node
(0,0,λ,λ)	(0,(0,λ),∅)
(1,0,λ,1)	(1,(0,1),((1,λ),(1,3-4)))
(2,2,λ,λ)	(2,(2,λ),∅)
(2,0,λ,1-2)	(2,(0,1-2),((2,λ),(2,3-4)))
(1,1,λ,λ)	(1,(1,λ),∅))
(2,2,λ,1-2)	(2,(2,1-2),((2,4)))
(2,1,λ,2)	(2,(1,2),((2,λ)))
(1,1,λ,1-2)	(1,(1,1-2),((2,λ)))
(2,1,λ,1-2-2)	(2,(1,1-2-2),((2,4)))
(1,1,λ,2)	-
(2,1,λ,2-2)	-

Table 5.3 - Comparison of Algorithm 5 Nodes and YK-Algorithm Nodes for P_1 in Protocol 5

There is a one-to-one correspondence between the nodes visited by Algorithm 5 and the nodes visited by the YK-algorithm, with the exception of two Algorithm 5 nodes: (1,1,λ,2), and (2,1,λ,2-2). The YK-algorithm does not generate nodes corresponding to (1,1,λ,2) and (2,1,λ,2-2). This is because expansion from (2,(1,1-2-2),((2,4))) is terminated, because (2,(1,1-2-2),((2,4))) is merged to (2,(2,1-2),((2,4))). This is caused by what Yuang and Kershenbaum call the "merge rule" [Yuan 89], i.e. the condition in line 1 of checkTerminal expressible as follows:

$$\exists\ V' \in A \text{ such that } ((s' = s) \text{ and } (Y \neq \varnothing) \text{ and } (Y' \supseteq Y))$$

This rule causes the YK-algorithm traversal to differ from the Algorithm 5 traversal. To see that it causes the YK-algorithm traversal to differ from a PEG traversal, consider Protocol 6:

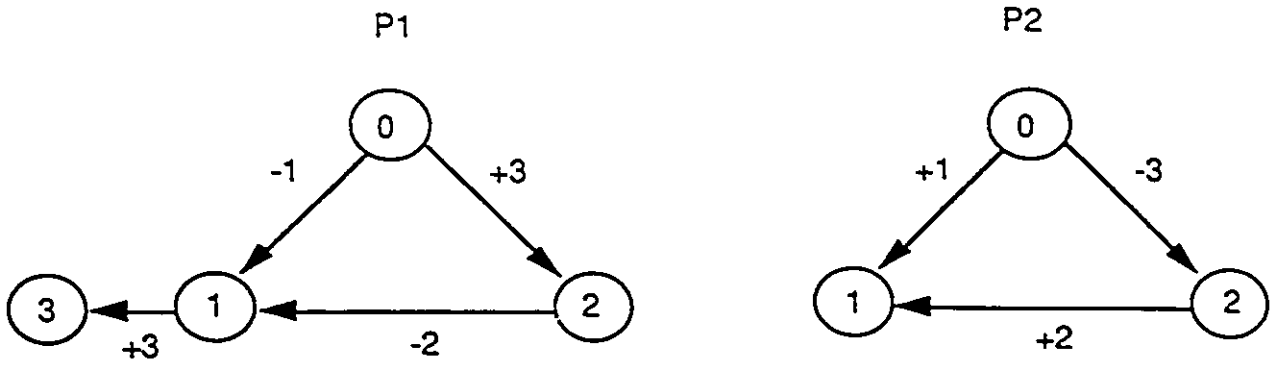

Figure 5.6 - Protocol 6

The RG for Protocol 6 is given in Figure 5.7:

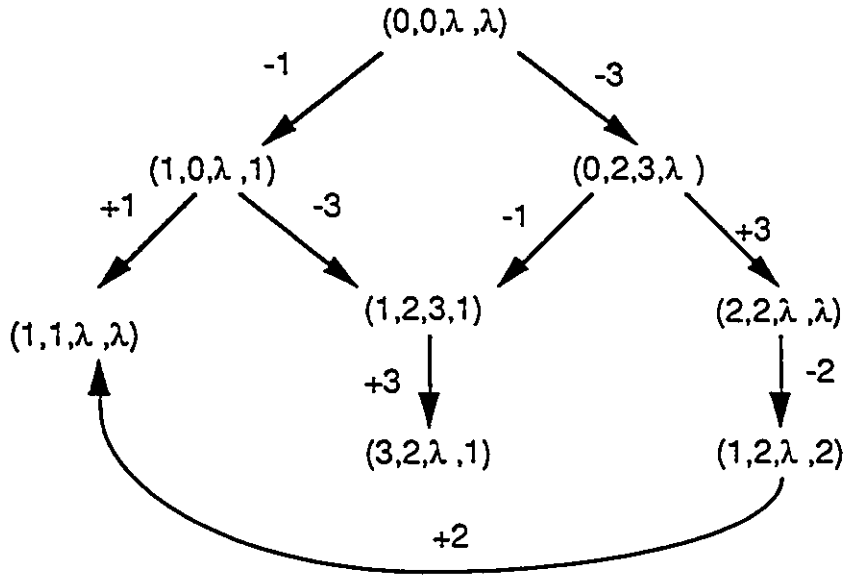

Figure 5.7 - Reachability Graph for Protocol 6

From (0,(0,λ),∅), the YK-algorithm would reach two nodes: (1,(0,1),(1,λ)) and (2,(2,λ),∅). From (1,(0,1),(1,λ)), one node would be reached: (3,(2,1),∅). From (2,(2,λ),∅), one node would be reached: (1,(2,2),(1,λ)). But (1,(2,2),(1,λ)) would be merged with (1,(0,1),(1,λ)). In the RG of Figure 5.7, this corresponds to drawing the -2 edge from (2,2,λ,λ) to (1,0,λ,1), instead of creating the (1,2,λ,2) node. If the YK-algorithm's output graph were to be interpreted as a PEG, this would imply that P_1 in

Protocol 6 can execute the sequence (+3,-2,+3), which it can not. Therefore the "merge rule" prevents the YK-algorithm from generating PEGs.

Of course, Yuang and Kershenbaum made no mention of language-based analysis in their paper [Yuan 89], and proposed the YK-algorithm solely for the purpose of detecting deadlocks, queue overflows, unspecified receptions, and unexecutable transitions.

6. CONCLUSIONS

A PEG describes a process language, i.e. the language consisting of the set of executable sequences of a process in a protocol. One way to generate a PEG, given a protocol and a specified host process, is by the tabular method, which requires that a reachability graph be generated first. Algorithm 4 improves on the tabular method by generating a PEG in a single phase, without generating a reachability graph. Algorithm 5 traverses fewer events than Algorithm 4 and, given a protocol in which no process receives from more than one other process, will either detect an error state or generate a PEG. A PEG generated by Algorithm 5 is called a PEG_5.

A process is said to be "effective" if all its specified sequences are executable. Given a PEG, effectiveness of the host process can be determined by minimizing both the PEG and the specification graph, and checking for an isomorphism between them. If the PEG and specification are deterministic, effectiveness can be determined in polynomial time. Specification graphs are deterministic by definition. For protocols consisting of two processes, where the non-host process has no state with both an outgoing send edge and an outgoing receive edge, PEG_5s are always deterministic. For other protocols, PEG_5s may be deterministic or non-deterministic.

In the general case, the problem of determining whether two FSMs accept the same language is PSPACE-complete, if either is non-deterministic [Hunt 76, Gare 79]. PEGs, however, represent a special case, since every PEG node is a final state. As future work, it would be interesting to derive complexity bounds for this case.

Every node of a PEG_5 maps to some reachable global state. Every reachable global state can be reached by an ε-sequence, i.e. a sequence of events generated by non-host processes, from a global state corresponding to some PEG_5 node. This fact can be used to introduce a degree of parallelism into reachability analysis, by generating a PEG_5, and performing a localized reachability analysis for each PEG node in parallel. The localized reachability analysis considers non-host events only, and uses the global state corresponding to the PEG_5 node as the initial state in each case.

Another application of PEGs is to detect process blockages, i.e. global states from which the host process may be unable to progress. A process blockage may be due to a non-host cycle, or the fact that there is an ε-sequence from that global state that leads to an error state (i.e. deadlock, queue overflow or unspecified reception). Process blockages

may be detected by first generating a PEG_5, and then executing Algorithm 6, which marks PEG_5 nodes that correspond to process blockage.

There are many other ways in which knowledge of the process' executable sequences could be used. The most general use would be simply to inspect the PEG to gain a better understanding of the protocol's behaviour. More particularly, a PEG could be used for the following:

1) determine whether the process' behaviour corresponds to some desired behaviour

2) identify particular specified sequences that are not executable

3) determine whether particular specified sequences are executable

4) generate local scenarios (i.e. scenarios involving the host process only)

5) replace the specification graph with a minimized PEG

Determining whether the PEG's behaviour corresponds to some desired behaviour is a generalized version of the problem of determining whether a process is effective. To do this, simply replace the specification graph by a graph describing the desired behaviour of the process, and take the approach described in section 4.1. This might be useful in the protocol synthesis strategy proposed by Rudie and Wonham [Rudi 90], in which a "supervisor" process is used to restrict the executable sequences of a "plant" process, in order to make the plant conform to some desired behaviour.

To detect particular specified sequences that are not executable, we could begin with the graph minimization strategy described in section 4.1. Given minimal graphs that are not isomorphic, we would know that some specified sequences are not executable, but a step would have to be added to detect such sequences. The simplest approach is to systematically compare specification sequences with PEG sequences, beginning with all sequences of length one and checking sequences of increasing length until an unexecutable sequence is found. Obviously the number of sequences that would have to be checked grows very quickly as the size of the specification grows. Nevertheless, a systematic search of fixed duration could check a large number of sequences, and might uncover an unexecutable specified sequence, and thereby be conclusive. If the systematic search was

inconclusive, we could, as a last resort, randomly select longer sequences for comparison. More intelligent strategies could undoubtedly be devised with further research.

Determining whether a particular specified sequence is executable is a simple, but potentially useful, application of a PEG, and needs no further explanation.

Generation of local scenarios, i.e. process event sequences, from a PEG is also a simple task, and might be used during a design review. The protocol designer could generate a number of such scenarios automatically, and then explain the logic behind each. Although global scenarios are also important during reviews, local scenarios are simpler, and often easier to understand.

Finally, a PEG could be minimized, and the resultant graph could be used to replace the process specification graph. The benefit would be that the specification would describe executable sequences only, and might therefore give a clearer representation of the protocol. Of course, in some cases a minimized PEG might obscure semantic relationships that existed in the original specification, and be more difficult to understand.

Two other algorithms, the maximal progress algorithm [Goud 84] and the YK-algorithm [Yuan 89], are superficially similar to Algorithm 5. The maximal progress algorithm was shown to generate a very different traversal of reachable states than is generated by Algorithm 5. The YK-algorithm generates a traversal that is similar to that of Algorithm 5, but which merges nodes not merged by Algorithm 5. Both the maximal progress algorithm and the YK-algorithm were written for the purpose of detecting error states, not for language-based analysis.

This thesis provides a basis for further research into language-based analysis of protocols. In particular, further study is required to adapt the given techniques to realistic protocols. Although Algorithm 5 does not suffer from the state space explosion problem to the same extent as reachability analysis, the astronomically large state spaces of some protocols [Holz 87] would obviously cause problems. Further improvements may be possible by incorporating elements of existing relief strategies. For example, it may be possible to employ a scatter search [Holz 87] in the step that detects the existence of sequences of type $H_S+N_O*N_SH_r$ from a given node.

APPENDIX A: GLOSSARY OF SYMBOLS AND ACRONYMS

· concatenation operator.

: restriction operator.

B queue bound.

D global state transition function.

D* iterated operation of D.

e element of E, i.e. an event.

E set of all events specified in a protocol.

+E set of all receive events specified in a protocol.

-E set of all send events specified in a protocol.

E_i union of $+E_i$ and $-E_i$, i.e. set of all events specified in P_i.

$+E_i$ set of receive events specified in P_i.

$-E_i$ set of send events specified in P_i.

E_i* set of all finite or infinite sequences constructed from members of E_i.

FIFO first-in, first-out.

FSM finite state machine.

G(X) reachability graph of protocol X.

H an event type, representing any host event.

H_r an event type, representing any receive by host process.

H_s an event type, representing any send by host process.

L(X,B) set of error-free or error-terminated executable event sequences of X, given bound B.

$L_i(X,B)$ set of error-free or error-terminated executable event sequences of P_i in X, given bound B.

$LM_i(X,B)$ set of maximal executable event sequences of P_i in X, given B.

m a message.

+m a receive event, i.e. the reception of message m.

-m a send event, i.e. the transmission of message m.

M set of all messages referenced in E.

M_i set of messages specified as sendable or receivable by P_i.

$+M_i$ set of messages specified as receivable by P_i.

$-M_i$ set of messages specified as sendable by P_i.

M^* set of all finite or infinite sequences of messages.

MPEG marked process event graph.

N_o receive by non-host process, or send from non-host to non-host.

N_r receive by non-host process.

N_s send to host process, by non-host process.

p a finite prefix of some sequence.

P n-tuple of all CFSMs in a protocol.

P_i element i from n-tuple P, i.e. a process.

PEG process event graph.

PEG_T PEG generated by the tabular method.

PEG_3 PEG generated by Algorithm 3.

PEG_4 PEG generated by Algorithm 4.

PEG_5 PEG generated by Algorithm 5.

Q n-tuple of all queues in a protocol.

Q_i element i from n-tuple Q, i.e. the input queue to P_i.

q_i contents of Q_i.

r initial state of an FSM.

R initial global state of a protocol.

R(X) set of all reachable global states of protocol X.

$R_B(X)$ set of reachable global states of protocol X with queue bound B.

RG reachability graph.

X a protocol.

V a global state of protocol X.

V′ a global state of protocol X.

V″ a global state of protocol X.

v_i current state of P_i.

α a sequence of events (or labels representing events).

$\alpha:E_i$ restriction of sequence α to event set E_i.

β a sequence of events (or labels representing events).

δ transition function of an FSM.

δ^* iterated operation of δ.

δ_i transition function of process P_i.

δ_i^* iterated operation of δ_i.

ε an event type, representing any non-host event.

ε-cycle a non-null ε-sequence that contains a cycle.

ε-sequence a (possibly null) sequence of events, each of type ε.

λ null sequence (of events or messages, depending on context).

σ a state of a process.

σ_i set of states for process P_i.

σ_{i0} initial state of process P_i.

BIBLIOGRAPHY

[Bran 83] D. Brand, P. Zafiropulo, "On Communicating Finite State Machines", JACM, Vol. 30, No. 2, April 1983.

[Clar 86] E.M. Clarke, E.A. Emerson, A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986.

[Gare 79] M.R. Garey, D.V. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness", W.H. Freeman and Co., 1979.

[Goud 84] M.G. Gouda, Y.T. Yu, "Protocol Validation by Maximal Progress State Exploration", IEEE Transactions on Communications, January 1984.

[Holz 87] G. Holzmann, "Automated Protocol Verification in Argos, Assertion Proving and Scatter Searching", IEEE Transactions on Software Engineering, June 1987.

[Holz 91] G. Holzmann, "Design and Validation of Computer Protocols", Prentice Hall, 1991.

[Hopc 79] J.E. Hopcroft, J.D. Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley, 1979.

[Hunt 76] H.B. Hunt III, D.J. Rosenkrantz, T.G. Szymanski, "On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages", Journal of Computer and System Sciences 12, 1976.

[Huus 91] J. Huus, H. Ural, "Validating Protocols by Generating Process Event Graphs", University of Ottawa Computer Science Dept. Technical Report TR-91-20, May 1991.

[Kaku 86] Y. Kakuda, Y. Wakahara, M. Norigoe, "A New Algorithm for Fast Protocol Validation", Proc. IEEE COMPSAC, 1986.

[Lin 87] F.J. Lin, P.M. Chu, M.T. Liu, "Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies", Computer Communications Review, Vol. 17, No. 5, 1987.

[Maxe 87] N. Maxemchuck, K. Sabnani, "Probabilistic Verification of Communication Protocols", Protocol Specification, Testing, and Verification VII, Elsevier Science Publishers, 1987.

[Okum 87] K. Okumura, "Protocol Analysis From Language Structure", Protocol Specification, Testing, and Verification VII, Elsevier Science Publishers, 1987.

[Rubi 82] J. Rubin, C.H. West, "An Improved Protocol Validation Technique", Computer Networks, April 1982.

[Rudi 90] K. Rudie, W.M. Wonham, "Supervisory Control of Communicating Processes", Protocol Specification, Testing, and Verification X, Elsevier Science Publishers, 1990.

[West 86] C.H. West, "Protocol Validation by Random State Space Exploration", Protocol Specification, Testing, and Verification VI, Elsevier Science Publishers, 1986.

[Yuan 88] M.C. Yuang, "Survey of Protocol Verification Techniques Based on Finite State Machine Models", Proc. NBS Computer Networking Symposium, 1988.

[Yuan 89] M.C. Yuang, A. Kershenbaum, "Parallel Protocol Verification: The Two-Phase Algorithm", Protocol Specification, Testing, and Verification IX, Elsevier Science Publishers, 1989.

[Zafi 78] P. Zafiropulo, "Protocol Validation by Duologue-Matrix Analysis", IEEE Transactions on Communications, Vol. 26, No. 8, August 1978.

[Zafi 80] P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan, D. Brand, "Towards Analyzing and Synthesizing Protocols", IEEE Transactions on Communications, April 1980.