Language-Based Analysis of Communicating Finite State Machines

Jan Huus
Simware, Inc.
Ottawa, Canada

Hasan Ural
University of Ottawa
Ottawa, Canada

Abstract

We present an efficient method for extracting a behavior description of a process (communicating finite state machine), as constrained by a given protocol. In the case where the constrained behavior corresponds to the unconstrained behavior depicted by the process specification, the process is said to be "effective". We call the derived behavior description a process event graph (PEG). By comparing the PEG with the process specification, we determine whether a process in a protocol is effective. Two extensions to the basic algorithm are given: one to introduce parallelism to reachability analysis, and one to detect traces that lead to process blockage. We focus initially on the two-process case, and then consider implications of more general protocols.

1: Introduction

Protocol validation has traditionally focused on detecting errors defined in terms of global states [3, 4, 11, 12]. Reachability analysis and variants such as supertrace [4] are symmetric, and treat every process equally. The maximal progress algorithm [3] and the YK-algorithm [11] are asymmetric, and analyze a protocol from the perspective of a specified process.

A complementary approach is to analyze the *protocol language*, defined as the set of *global traces* (executable global event sequences) of a protocol, or the *process language*, defined as the set of *local traces* (executable event sequences of a process in a protocol). Okumura [9] analyzed the general properties of protocol languages and process languages in the *communicating finite state machine* (CFSM) model, but did not examine methods of deriving a process language. In this paper, we present an efficient method for deriving the process language of a specified process in a protocol. Like the maximal progress algorithm and YK-algorithm, we take an asymmetric approach. Unlike those algorithms, we are interested in deriving the process language.

The CFSM model is well-known. Briefly, a process (CFSM) is a finite state machine in which each transition is either a *send event* or a *receive event*. Asynchronous communication among processes is via simplex channels, modelled by FIFO queues. Messages exchanged between processes are denoted by positive integers. A send of message n is denoted by -n, and a receive of n is denoted by +n. For protocols consisting of exactly two processes, no destination need be specified, since processes may not send to themselves.

For every process P_i in protocol X, we define the *event set* E_i to be the set of all specified send and receive events of P_i. For nodes v and w in the specification graph of P_i, and $e \in E_i$, we define $\delta_i(v,e) = w$ iff there is an edge labelled e from v to w. A *global state* $V = (v_1, ..., v_n, q_1, ..., q_n)$ of protocol X is a vector containing the current state v_i of each process, and the contents q_i of each input queue. In the initial global state, V_0, all processes are in their initial states, and all queues are empty. The set of all global states reachable from V_0 is called the *reachable global state space*, R(X). By declaring a finite bound B on the number of messages in any queue, we can ensure that R(X) is finite. Any global state V in which a queue length exceeds B is then declared an *error state*, and no transitions are considered to be executable from V. Error states may also be declared in other cases, such as deadlock.

The graph defined by creating a node for every reachable global state of protocol X, and creating a directed edge from V_i to V_j for every executable transition from V_i to V_j is called the *reachability graph*, G(X,B), where B is the queue bound. The queue bound affects the reachability graph since it determines error states, from which no transitions are considered possible. Each edge e in G(X,B) is labelled with the corresponding event, i.e. +n for a receive, and -n for a send. The *global state transition function* for X is the partial function D such that $D(V,e) = V'$ iff there is an edge labelled e from V to V' in G(X,B). If there is a sequence of directed edges beginning at V and ending at V', such that the labels of the edges traversed form the sequence α, we say there is a trace labelled α from V to V'. D^* is the iterated

0-8186-3670-X/93 $03.00

operation of D, i.e. $D^*(V,\alpha) = V'$ iff there is a trace labelled α from V to V' in G(X,B). If there is no trace labelled α from V, $D^*(V,\alpha)$ is undefined. For convenience, $D^*(V_0,\alpha)$ is abbreviated to $D^*(\alpha)$, where V_0 is the initial global state.

The *restriction* of trace α to event set E_i, denoted $\alpha{:}E_i$, is defined as follows, where λ denotes the null trace and $\alpha{\cdot}e$ represents the concatenation of α and e:

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 $e \notin E_i$

A *global trace* α, given protocol X with queue bound B, is a trace α such that $D^*(\alpha)$ is defined. The *protocol language* of protocol X, denoted L(X,B), is the set of all global traces. A *local trace* β of process P_i in protocol X is a trace β such that $\beta = \alpha{:}E_i$ for some global trace α. The *process language* of process P_i, denoted $L_i(X,B)$, is the set of all local traces. 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 traces beginning at r in G is identical to $L_i(X,B)$.

An ε-*move* of an FSM is a spontaneous state transition, i.e. a transition with no input. An ε-*sequence* is a (possibly null) sequence of ε-moves. An ε-*cycle* is an ε-sequence that contains a cycle.

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

error_state(V,B) = overflow(V,B) or deadlock(V)
overflow(V,B) = TRUE iff $\exists\ i \in [1,n]$ s.t. $|q_i| > B$
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

We have chosen a simple error_state function for ease of exposition, but a more complex function could be defined.

2: Deriving a Process Event Graph

The process for which a PEG is to be derived will be called the *host* process, and the other process will be called the *non-host* process. Taking the perspective of classical language theory [6], we could view the reachability graph, G(X,B), as an FSM, and view all non-host edges (i.e. transitions executed by the non-host process) as ε-moves. If every global state were treated as a final state, G(X,B) with ε-moves would accept $L_i(X,B)$. It is shown in [6] that FSMs with ε-moves can be converted to equivalent FSMs without ε-moves. This approach can be used to generate a PEG from G(X,B). In [6], a tabular method is given. The tabular method can be summarized as follows:

1) Create a state/event table with one row per global state, one column per host event, and an extra column for ε.
2) In every state/event table entry, excluding the ε column, list the states that are reachable from the corresponding state by some trace consisting of an ε-sequence followed by a single host event, followed by an ε-sequence (such a trace may be said to be of type $\varepsilon^*H\varepsilon^*$, where an event of type H is a host event).
3) In every row of the ε column, list the states that are reachable from the corresponding state by an ε-sequence.
4) 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.
5) Mark final states as follows:
- every state that was final in the original graph (with ε-moves) is also final in the resultant graph (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.

Since every state of G(X,B) is considered as final for our purposes, step 5 of the tabular method can be omitted. It would be desirable to generate the PEG in a single pass, however, instead of first generating G(X,B). A first attempt to combine the tabular method with reachability analysis might be to explore all traces of type $\varepsilon^*H\varepsilon^*$ from each global state S, when S is being expanded. This approach has the following problems:

1) Redundant PEG edges would be introduced when multiple traces of type $\varepsilon^*H\varepsilon^*$ from s_1 to s_2 have the same host event.
2) The number of traces of type $\varepsilon^*H\varepsilon^*$ from a given node S could be infinite, in the case of ε-cycles.
3) Intermediate global states traversed by a trace of type $\varepsilon^*H\varepsilon^*$ must be examined for errors, to ensure that the trace corresponds to a sequence in G(X,B).

These problems can be resolved by adding a check for redundant edges, and by searching only those traces of type ε*Hε* that are cycle-free and either error-free or error-terminated. With these additions, we derive Algorithm 1, shown below. Input parameters are as follows:

- X: input protocol
- B: queue bound
- i: subscript to identify host process

```
genPEG(X, B, i)
1.   R := (σ10, ..., σn0, λ, ..., λ);
2.   W := R;
3.   A := ∅;
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 trace 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 the 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 1: PEG Generation During Reachability Analysis

Algorithm 1 assumes the existence of an error_state function, as defined in section 1. The algorithm traverses at least part of the reachable global state space of protocol X with bound B, and constructs a PEG as it proceeds. However, it can be shown that the PEG generated by Algorithm 1 is much larger than necessary. For example, consider Protocol 1:

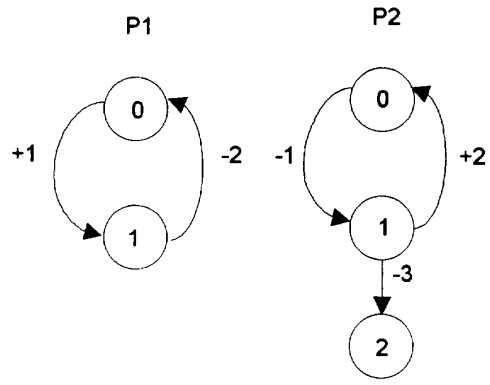

Figure 2.1: Protocol 1

The reachability graph for Protocol 1 is as follows:

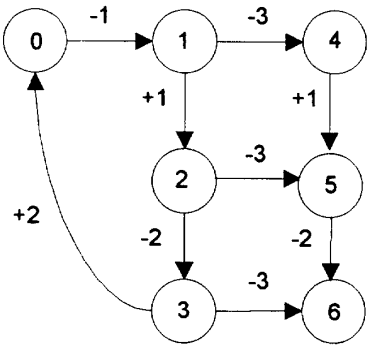

Figure 2.2: Reachability Graph for Protocol 1

Note that the reachability graph given in Figure 2.2 corresponds to B > 1. The PEG generated by Algorithm 1 for process P_1 in Protocol 1 is as follows:

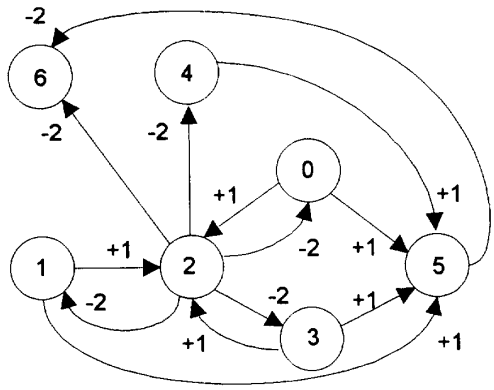

Figure 2.3: PEG Generated by Algorithm 1 for P_1 in Protocol 1

3: Omitting redundant shuffles during PEG derivation

PEGs generated by Algorithm 1 are much larger than necessary. This can be seen by comparing the PEG in Figure 2.3 with the specification graph of P_1, in Figure 2.1. The specification graph for P_1 can be shown to be a PEG, since all specified traces are executable. The excessive size of PEGs generated by Algorithm 1 arises from the fact that it traverses more traces than required in G(X,B). More efficient PEG generation can be achieved by avoiding redundant shuffles, i.e. sequences in which the local traces remain constant, but the global interleaving changes. This is similar to the approach taken in [5], except that they were interested in preserving local and global correctness properties, while we are interested only in preserving local traces.

Redundant shuffles can be avoided by pruning the set of edges traversed from a given global state, but we must be able to prove that the method of pruning preserves local traces. That is, by pruning an edge we neither increase nor decrease the potential traces of the host process. We will begin by considering error-free two-process protocols only. More general protocols will be examined once we have derived results for this basic case.

Consider a refinement of the event classification into four categories:

H_S - send by host
H_r - receive by host
N_S - send by non-host
N_r - receive by non-host

The traces of type $\varepsilon^*H\varepsilon^*$ that are traversed by Algorithm 1 are of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$ using the refined event classification.

Theorem 3.1: Given a global state V in an error-free two-process protocol X, such that the host input queue is empty at V, and $D^*(V,\alpha)$ is defined for some trace α of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$, there exists a trace β of type $(H_S+N_r{}^*N_SH_r)(N_S+N_r)^*$ such that $D^*(V,\alpha) = D^*(V,\beta)$.

Proof: A trace α of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$ is either a host send trace of type $(N_S+N_r)^*(H_S)(N_S+N_r)^*$ or a host receive trace of type $(N_S+N_r)^*(H_r)(N_S+N_r)^*$. Consider a host send trace $\alpha = \alpha_1{\cdot}\alpha_2{\cdot}\alpha_3$, where α_1 is the $(N_S+N_r)^*$ prefix, α_2 is the H_S event, and α_3 is the $(N_S+N_r)^*$ suffix. The executability of α_2 depends only on the host process state, which is not affected by α_1. Therefore α_2 may be executed before α_1. The executability of α_1 can be disabled by prior execution of α_2 only if α_2 generates an error state, but the protocol is assumed to be error-free. Therefore $\beta = \alpha_2{\cdot}\alpha_1{\cdot}\alpha_3$ must also be executable from V, and $D^*(V,\alpha) = D^*(V,\beta)$.

Now consider a host receive trace $\alpha = \alpha_1{\cdot}\alpha_2{\cdot}\alpha_3$, where α_1 is the $(N_S+N_r)^*$ prefix, α_2 is the H_r event, and α_3 is the $(N_S+N_r)^*$ suffix. Since the host input queue is empty at V, α_1 must contain at least one N_S event; otherwise α_2 would not be executable. So $\alpha_1 = \alpha_{11}{\cdot}\alpha_{12}{\cdot}\alpha_{13}$, where α_{11} is of type $N_r{}^*$, α_{12} is of type N_S, and α_{13} is of type $(N_S+N_r)^*$. Clearly α_2 is executable after $\alpha_{11}{\cdot}\alpha_{12}$, and the execution of α_2 does not affect the executability of α_{13}. Therefore $\beta = \alpha_{11}{\cdot}\alpha_{12}{\cdot}\alpha_2{\cdot}\alpha_{13}{\cdot}\alpha_3$ must also be executable from V, and $D^*(V,\alpha) = D^*(V,\beta)$.

Combining these two cases, we can see that, from a given global state V in which the host input queue is empty, executability of a trace α of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$ implies executability of a trace β of type $(H_S+N_r{}^*N_SH_r)(N_S+N_r)^*$ such that $D^*(V,\alpha) = D^*(V,\beta)$.
QED.

Theorem 3.2: Given a global state V in an error-free two-process protocol X, such that the host input queue is empty at V, and $D^*(V,\alpha{\cdot}\alpha')$ is defined for traces α and α' of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$, there exists a trace $\beta{\cdot}\beta'$ such that β is of type $H_S+N_r{}^*N_SH_r$, β' is of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$, and $D^*(V,\alpha{\cdot}\alpha') = D^*(V,\beta{\cdot}\beta')$.

Proof: From Theorem 3.1, there is an executable trace $\beta{\cdot}\beta''$ of type $(H_S+N_r{}^*N_SH_r)(N_S+N_r)^*$ such that β is of type $H_S+N_r{}^*N_SH_r$, β'' is of type $(N_S+N_r)^*$, and $D^*(V,\beta{\cdot}\beta'') = D^*(V,\alpha)$. Therefore $\beta' = \beta''{\cdot}\alpha'$ is of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$ and $D^*(V,\alpha{\cdot}\alpha') = D^*(V,\beta{\cdot}\beta')$ as required.
QED.

In the initial state of a protocol, the host input queue is empty. Algorithm 1 traverses every trace of type $(N_S+N_r)^*(H_S+H_r)(N_S+N_r)^*$ from the initial state. Theorem 3.2 shows that it suffices to traverse traces of type $H_S+N_r^*N_SH_r$, since each such trace leaves the host input queue empty and, by repeated application of Theorem 3.2, ensures that no host event will be missed.

Therefore, we modify Algorithm 1 and obtain Algorithm 2 by traversing only those traces of type $H_S+N_r^*N_SH_r$, instead of all traces of type $\varepsilon^*H\varepsilon^*$. To examine the improvement achieved by Algorithm 2, we will apply it to Protocol 1, shown in Figure 2.1 above. For clarity, we first show the reachability graph with event types added:

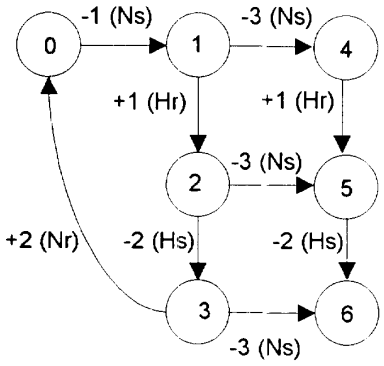

Figure 3.1: Reachability Graph for Protocol 1 with Event Types Shown

Applied to Protocol 1, Algorithm 2 proceeds as follows:

1) Set A empty and store node 0 (the initial state) 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 trace of type $N_r^*N_SH_r$ (-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 trace of type $N_r^*N_SH_r$. 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 trace of type $N_r^*N_SH_r$, (+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 3.2:

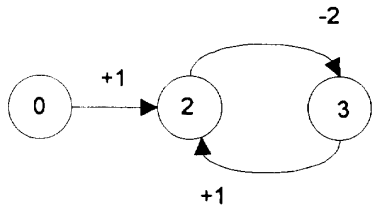

Figure 3.2: PEG Generated by Algorithm 2 for P_1 in Protocol 1

The PEG produced by Algorithm 2 (Figure 3.2) is much smaller than the PEG produced by Algorithm 1 (Figure 2.3). Where it is necessary to distinguish the two, a PEG generated by Algorithm 1 will be called a PEG_1, and a PEG generated by Algorithm 2 will be called a PEG_2.

4: Applications of Process Event Graphs

4.1: Validating process effectiveness

As noted in the introduction, a process is said to be *effective* in a protocol if every specified local trace is executable [9]. One way to determine whether all specified traces are executable is to determine whether the PEG and specification graph accept the same language. In our CFSM model, a process corresponds to a deterministic FSM. That is, given event e and process 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 [2, 7]. It is therefore interesting to investigate the conditions under which PEGs are deterministic.

Theorem 4.1: For any two-process protocol in which no state of the non-host process has both an outgoing send edge and an outgoing receive edge, the PEG_2 is deterministic.

Proof: The host input queue is empty initially, and remains empty at the end of every trace of type $H_S+N_r^*N_SH_r$. For a PEG_2 to be non-deterministic, therefore, there must exist some global state V in which the host input queue is empty, and two traces α and α' of type $H_S+N_r^*N_SH_r$ such that $D^*(V,\alpha) \neq D^*(V,\alpha')$, where the host event in α is identical to the host event in α'.

First, consider traces α and α' of type H_S. Since the host event in α is identical to the host event in α', $\alpha = \alpha'$. Since the specification graphs are deterministic, $D^*(V,\alpha) = D^*(V,\alpha')$.

Now consider traces α and α' of type $N_r^* N_S H_r$. If $\alpha \neq \alpha'$, but the host event in α is identical to the host event in α', there are two possibilities:

1) the length of the N_r^* prefix is different in α and α'
2) the N_S event is different in α and α'

Note that the N_r^* prefixes cannot be different but of the same length, since the sequence of non-host receives is determined by the contents of the non-host input queue at V. In case (1), one N_r^* prefix is shorter than the other. But then the trace with the shorter N_r^* 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. 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 interesting 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_2 will be non-deterministic. As an example, consider Protocol 2, from [3]:

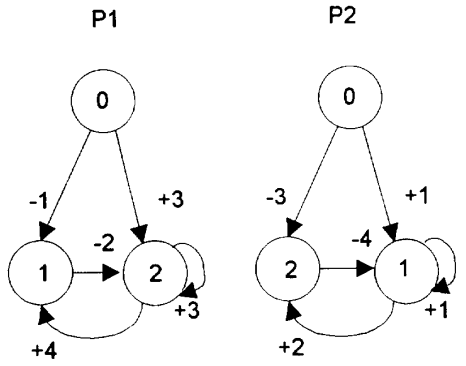

Figure 4.1: Protocol 2

The traversal of Protocol 2 by Algorithm 2, with P_1 as host is shown in Figure 4.2 below. Global states are shown in the form (s_1, s_2, q_1, q_2), where s_i is the state of process P_i, and q_i represents the contents of the input queue for P_i. The empty queue is shown as λ, and multiple messages are shown as concatenated digits, with the head of the queue in the leftmost position. The trace traversed for each edge generated is also shown.

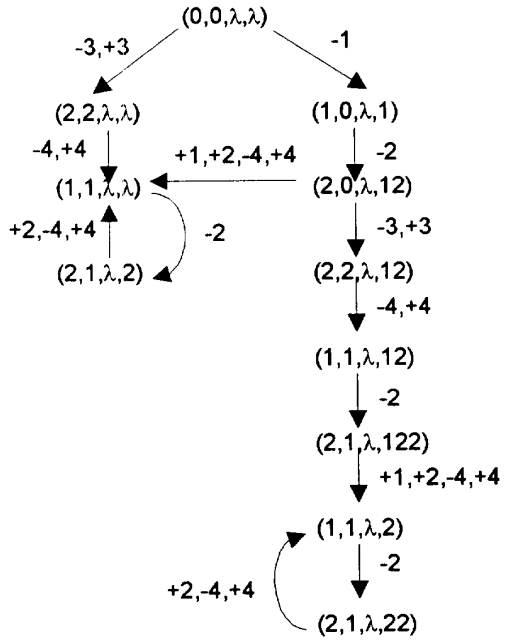

Figure 4.2: Protocol 2 Traversal by Algorithm 2 with P_1 as Host

To obtain a clearer representation of the PEG_2 for P_1 in Protocol 2, the graph in Figure 4.2 was minimized, using a variation of the procedure described in [6]. A variation is needed since the algorithm in [6] assumes that the set of final states is a strict subset of the states of the FSM. This is used to identify pairs of states that can be distinguished in the initial step. In our case, we distinguish states p and q in the initial step iff there exists a sequence α such that exactly one of $\delta^*(p,\alpha)$ and $\delta^*(q,\alpha)$ is defined. The resultant PEG is given in Figure 4.3:

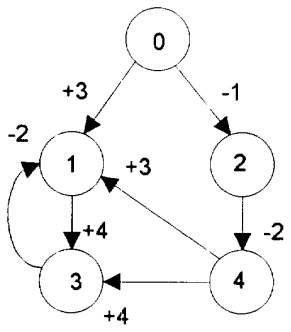

Figure 4.3: Minimized PEG_2 for P1 in Protocol 2

State 0 of P_2 in Protocol 2 has an outgoing send edge and an outgoing receive edge, but, as shown in Figure 4.3, the PEG for P_1 is deterministic.

Hopcroft et al state that, for a given language L, the minimum deterministic FSM accepting L is unique up to an isomorphism [6]. Since the minimal PEG for P_1 in Protocol 2 has more nodes than the specification graph, they are not isomorphic, so there must be unexecutable specified traces. That is, P_1 is not effective in Protocol 2. Beginning a systematic search of specified traces of increasing length, we discover an unexecutable specified trace of length 2: (+3,+3). At a more general level, the PEG reveals that P_1 will eventually enter a (+4,-2) loop with no exit, and there are three traces that lead to it: (+3), (-1,-2), and (-1,-2,+3). In addition to determining that P_1 is not effective, this example shows that the PEG provides information about the behaviour of the protocol that is not readily apparent by direct inspection of the specification.

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_2, and conducting a separate partial reachability analysis from each PEG_2 node.

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

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

Theorem 4.3: Given an error-free two-process protocol X, every global state V reachable by a trace α of type $(\varepsilon^*H\varepsilon^*)^*$ is reachable by an ε-sequence from some global state V' that is reachable by a trace β of type $(H_S+N_r^*N_SH_r)^*$.

Proof: Theorem 3.1 proved that, if V is reachable by a trace α of type $\varepsilon^*H\varepsilon^*$, then it is reachable by a trace β of type $(H_S+N_r^*N_SH_r)\varepsilon^*$. Therefore, if V is reachable by a trace α' of type $(\varepsilon^*H\varepsilon^*)(\varepsilon^*H\varepsilon^*)$, it must be reachable by a trace β' of type $(H_S+N_r^*N_SH_r)\varepsilon^*(\varepsilon^*H\varepsilon^*)$. But then β' is of type $(H_S+N_r^*N_SH_r)(\varepsilon^*H\varepsilon^*)$, so V must be reachable by a trace of type $(H_S+N_r^*N_SH_r)(H_S+N_r^*N_SH_r)\varepsilon^*$. By induction, therefore, we can derive our result.
QED.

By Theorem 4.2, every reachable global state can be reached from the initial state either by an ε-sequence or by a trace of type $(\varepsilon^*H\varepsilon^*)^*$. By Theorem 4.3, every state that is reachable by a trace of type $(\varepsilon^*H\varepsilon^*)^*$ is reachable by an ε-sequence from a state that is reachable by a trace of type $(H_S+N_r^*N_SH_r)^*$. But every global state that is reachable by a trace of type $(H_S+N_r^*N_SH_r)^*$ is traversed by Algorithm 2, and corresponds to a node in a PEG_2. The initial state also corresponds to a node in a PEG_2. Therefore, every reachable global state can be reached by an ε-sequence from some global state corresponding to a PEG_2 node.

This allows us to introduce an element of parallelism to reachability analysis, by performing a local reachability analysis for each PEG_2 node as soon as it is generated, in parallel with Algorithm 2 and the other local analyses. Each local analysis begins with the corresponding PEG_2 node as its initial state, and traverses ε-sequences only. This approach provides a significant memory saving compared to traditional reachability analysis, since Algorithm 2 stores only those global states that are reachable by a trace of type $(H_S+N_r^*N_SH_r)^*$. For a comparison of the time utilization, however, further study is needed.

4.3: Detecting process blockage

We may define a *maximal local trace* for process P_i in protocol X with bound B, as a finite-length local trace α, such that $\alpha = \beta{:}E_i$ for some global trace β, and one of the following conditions is true:

1) error_state($D^*(\beta)$,B) = TRUE
2) there is an ε-sequence that begins at $D^*(\beta)$ and ends at an error state
3) there is an ε-cycle that begins at $D^*(\beta)$

That is, for every maximal local trace α, there is some global trace in which α may not be extended to a longer

local trace. There may, however, be another global trace in which α is not maximal. To see that this is possible, consider Protocol 3:

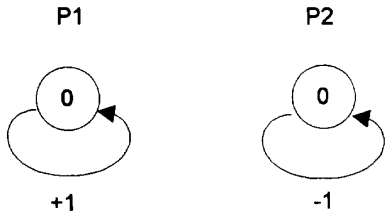

Figure 4.4: Protocol 3

With P_1 as host in Protocol 3, and B = 1, the local trace (+1) is maximal, because it is the restriction of (-1,+1,-1,-1), which generates an error state. On the other hand, (+1) may be extended to (+1,+1), by virtue of the global trace (-1,+1,-1,+1).

Let us define the *maximal process language* for process P_i in protocol X with queue bound B to be the set of all maximal local traces of P_i. The maximal process language may be identified by marking nodes of the PEG that correspond to global states for which one of the three conditions defined above are true. Each marked node corresponds to a potential *process blockage*, i.e. a global state from which the host process may be unable to progress.

4.4: Other applications of PEGs

There are many other ways in which knowledge of the process' executable traces could be used. The most general use would be simply to inspect the PEG to gain insight into the protocol's behaviour. For example, the PEG presented in Figure 4.3 differs dramatically from the corresponding specification graph in Figure 4.1, and gives a clearer picture of the actual behaviour of Protocol 2. Also, a PEG could be used for the following:

1) determine whether the process' behaviour corresponds to some desired behaviour
2) identify particular specified traces that are not executable
3) determine whether particular specified traces are executable
4) replace the specification graph with a minimized PEG

Determining whether the process' behaviour corresponds to some desired behaviour is a generalized version 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 [10], in which a "supervisor" process is used to restrict the behaviour of a "plant" process, in order to make the plant conform to some desired behaviour.

To detect particular specified traces that are not executable, we could begin by minimizing the specification graph and PEG. Given minimal graphs that are not isomorphic, we would know that some specified trace was not executable, but a step would have to be added to detect such traces. The simplest approach is to systematically compare specification traces with PEG traces, beginning with all sequences of length one, and checking sequences of increasing length until an unexecutable specified trace is found. If this search was inconclusive after some fixed duration, we could begin to randomly select longer sequences for comparison. More intelligent strategies could undoubtedly be devised with further research.

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

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 traces 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.

5: Arbitrary protocols

5.1: Error states

We have shown that Algorithm 2 generates a correct PEG in the absence of error states, and we will now consider the impact of error states. Let "omitted traces" denote global traces that are not traversed by Algorithm 2, and let "executed traces" denote global traces that are traversed by Algorithm 2. On first examination, there appear to be two cases in which error states could cause Algorithm 2 to fail to generate a correct PEG: an omitted trace contains an error that does not occur in the corresponding executed trace (omitted error), or an executed trace contains an error that does not occur in one of the corresponding omitted traces (reached error).

To see that omitted errors are possible, consider Protocol 3 again (Figure 4.4). With P_1 as host, Algorithm 2 traverses the trace (-1,+1) from the initial state back to the initial state and then terminates. If we

assume a queue bound of 1, reachability analysis will detect a queue overflow resulting from the trace (-1,-1), but Algorithm 2 will detect no error states. However, the definition of a PEG, given above, requires only that every executable trace of the process specification graph be a specified trace of the PEG, and vice versa. This is not compromised by omitted errors. That is, omitted errors may exist, but do not prevent Algorithm 2 from generating a correct PEG.

In the case of reached errors, Algorithm 2 may be prevented from executing host events that are executed by Algorithm 1. This would seem to be more serious than the case of omitted errors, since reached errors may cause Algorithm 2 to fail to generate a correct PEG. Because the error is identified, however, it may be corrected. Once the protocol is sufficiently correct that no error states are detected by Algorithm 2, a correct PEG will be generated.

Recall that our parallel approach to reachability analysis extends Algorithm 2 by traversing ε-sequences from every PEG_2 node. Error states that are not detected by Algorithm 2 will remain undetected by the parallel reachability analysis only if some error-terminated trace of type $(\varepsilon^*H\varepsilon^*)^*$ is not reachable by an ε-sequence from some PEG_2 node. But as long as all traces of type $(H_S+N_r^*N_SH_r)^*$ are error-free, Theorem 4.3 holds, so all error states are reachable. Therefore the parallel reachability analysis will detect all error states, if none are detected by Algorithm 2. If Algorithm 2 does detect errors, then some other error states may remain undetected. As discussed above, this is not a serious concern.

The same arguments apply to detection of process blockage. If Algorithm 2 detects no error states, all process blockages will be detected. Otherwise, some process blockages may remain undetected.

5.2: Multi-process protocols

Thus far, we have restricted our attention to protocols consisting of exactly two processes. Protocols with arbitrary numbers of processes may be defined according to one of two models: the *single-queue model*, in which every process has a single input queue shared by all senders (as in SDL [1]), and the *multi-queue model*, in which each process has one input queue for every sender (as in [9]). Of course, hybrid models may also be defined, in which case characteristics of both the single-queue and multi-queue models will be present.

To account for multi-process models, we must modify our event classification to include the possibility of a send from a non-host process to another non-host process. This may be done by replacing N_r with N_O ("other", i.e. receive by non-host or send from non-host to non-host). Let us suppose that we adapt Algorithm 2 to the multi-process case by traversing traces of type $H_S+N_O^*N_SH_r$ instead of traces of type $H_S+N_r^*N_SH_r$.

In the single-queue model, this leads to a problem in the generalization of Theorem 3.1, for the case of host send traces. Consider a host send trace $\alpha = \alpha_1\cdot\alpha_2\cdot\alpha_3$, where α_1 is the $(N_S+N_O)^*$ prefix, α_2 is the H_S event, and α_3 is the $(N_S+N_O)^*$ suffix. In the two-process case, α_2 may be executed ahead of α_1, without affecting the executability of α_1. In the multi-process single-queue model, however, α_2 might "disable" the executability of α_1 by inserting a message on a queue ahead of a message received in α_1. This problem can be avoided by traversing a broader set of traces, i.e. all traces of type $(N_S+N_O)^*\ H_S+N_O^*N_SH_r$.

In the multi-queue model, host send traces do not cause a problem. Host receive traces do, however, since the H_r event may not correspond to the first N_S event. For example, an $N_SN_SH_r$ trace may not have an executable $N_SH_rN_S$ shuffle, if the H_r event corresponds to the second N_S event. This problem can be avoided by traversing a broader set of traces, i.e. all traces of type $H_S+(N_S+N_O)^*H_r$.

6: Summary and conclusions

A PEG describes a process language, i.e. the language consisting of the set of executable traces of a process in a protocol. One way to generate a PEG, given a protocol and specified host process, is by the tabular method, which requires that a reachability graph be generated first. Algorithm 1 improves on the tabular method by generating a PEG in a single pass, without first generating a reachability graph. Algorithm 2 traverses fewer transitions than Algorithm 1. When the distinction is required, a PEG generated by Algorithm 1 is called a PEG_1, and a PEG generated by Algorithm 2 is called a PEG_2.

A process is said to be *effective* if all its specified traces are executable. If the PEG and specification are deterministic, effectiveness can be determined in polynomial time by minimizing both the PEG and the specification graph, and checking for an isomorphism between them. The problem of determining whether two FSMs accept the same language is PSPACE-complete, if either is non-deterministic [2, 7]. Specification graphs are deterministic by definition. For two-process protocols in which the non-host process has no state with both an outgoing send edge and an outgoing receive edge, PEG_2s are always deterministic. For more general protocols, PEG_2s may be deterministic or non-deterministic.

Every node of a PEG_2 maps to some reachable global state. Every reachable global state can be reached by an ε-sequence, i.e. a sequence of non-host events, from a global state corresponding to some PEG_2 node. This fact can be used to introduce a degree of parallelism to reachability analysis, by generating a PEG_2, and performing a local reachability analysis for each PEG_2 node in parallel. The local reachability analyses consider non-host events only, and use the global state corresponding to the PEG_2 node as the initial state in each case. For maximum parallelism, the local reachability analyses may begin as soon as a PEG_2 node is generated, rather than waiting for construction of the complete PEG_2.

Another application of PEGs is to detect process blockage, i.e. global states from which the host process may be unable to progress. A process blockage may be due to an ε-cycle, or the existence of an ε-sequence that leads to an error state. Local traces that lead to process blockage may be identified by first generating a PEG_2, and then traversing the PEG_2 in a second phase, marking nodes that correspond to process blockage. Other applications that were proposed are as follows:

1) provide insight into the protocol's behaviour
2) determine whether the process' behaviour corresponds to some desired behaviour
3) identify particular specified traces that are not executable
4) determine whether particular specified traces are executable
5) replace the specification graph with a minimized PEG

7: Bibliography

[1] F. Belina, D. Hogrefe, A. Sarma, "SDL with Applications from Protocol Specification", Prentice Hall, 1991.

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

[3] M.G. Gouda, Y.T. Yu, "Protocol Validation by Maximal Progress State Exploration", IEEE Transactions on Communications, Vol. COM-32, No. 1, January 1984.

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

[5] G.J. Holzmann, P. Godefroid, D. Pirottin, "Coverage Preserving Reduction Strategies for Reachability Analysis", Proceedings of the 12th Symposium on Protocol Specification, Testing, and Verification, Elsevier Science Publishers, 1992.

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

[7] H.B. Hunt, 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, Academic Press, Inc., 1976.

[8] J. Huus, "Language-Based Analysis of Communicating Finite State Machines", Master's Thesis, Department of Computer Science, University of Ottawa, 1992.

[9] K. Okumura, "Protocol Analysis from Language Structure", Proceedings of the 8th Symposium on Protocol Specification, Testing, and Verification, Elsevier Science Publishers, 1988.

[10] K. Rudie, W.M. Wonham, "Supervisory Control of Communicating Processes", Proceedings of the 10th Symposium on Protocol Specification, Testing, and Verification, Elsevier Science Publishers, 1990.

[11] M.C. Yuang, A. Kershenbaum, "Parallel Protocol Verification: The Two-Phase Algorithm", Proceedings of the 9th Symposium on Protocol Specification, Testing, and Verification, Elsevier Science Publishers, 1989.

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