An Incremental Protocol Verification Method

CHUNG-MING HUANG AND JENQ-MUH HSU

Institute of Information Engineering, National Cheng Kung University, Tainan, Taiwan 70101, ROC

Protocol verification is an activity to assure the correctness of communication protocols. Global state reachability analysis is one of the most straightforward and easily automated protocol verification methods. This paper proposes an incremental protocol verification method for the Extended Communicating Finite State Machine (ECFSM) model. Incremental protocol verification allows protocols to be modified at the run time of global state reachability analysis. Then, instead of re-exploring the modified protocols from scratch, global state reachability analysis is continued incrementally at the modification point. To enhance the efficiency, the proposed method incorporates the dead and live variables concept that is used in Chu and Liu's global state reduction technique (Chu and Liu, 1989). Using the proposed incremental protocol verification method, incremental protocol design environments are achievable for ECFSM-based Formal Description Techniques (FDTs), e.g. ISO's Estelle. Our application of the proposed method to Estelle is also briefly introduced in this paper.

Received January 31, 1994, revised August 26, 1994

1. INTRODUCTION

Being the backbone of distributed systems and computer networks, a communication protocol is a set of rules that governs the interactions among the communication entities (Bochmann, 1989; Linn, 1989; Liu, 1989; Pehrson, 1989; Sidhu, 1989b). In order to assure a protocol be free from logical errors, a complex and repeated cycle consisting of re-specification and re-verification is executed until there is no logical error in the protocol. In the past decade, many formal methods have been used to formally specify communication protocols. Many of these methods are based on the state transition model, such as the Communicating Finite State Machines (CFSM) model (Brand and Zafiropulo, 1989), petri nets (Diaz, 1982) and formal grammars (Anderson and Landweber, 1984; Umbaugh *et al.*, 1983).

Protocol verification is an activity to detect logical errors, such as deadlock, unspecified receptions and channel overflow (when communication channels are finite), in communication protocols (West, 1992; Gouda, 1993; Horlzmann, 1993). To have automatic protocol verification, many formal protocol verification methods were proposed in the past decade. Global state reachability analysis is one of the most straightforward ways to automatically detect logical errors in communication protocols specified in the state transition model (Lin *et al.*, 1987; Chanson *et al.*, 1993). Using global state reachability analysis, a global state reachability graph containing all possible transition sequences and all reachable global states in the communication protocol is generated.

Although global state reachability analysis is easily automated, global state reachability analysis suffers from the global state explosion problem. In order to relieve the state explosion problem, many state reduction techniques (Ioth and Ichikawa, 1983; Gouda, 1984, 1985; Lin *et al.*, 1987) and various protocol verification methods (Frieder and Herman, 1989; Huang *et al.*, 1990; Frieder, 1992) have been proposed in the past decade. Unfortunately, very few of these reduction techniques and various verification methods are based on the ECFSM model. Chu and Liu's reduction technique is one of the few reduction techniques which are based on the ECFSM model (Chu and Liu, 1989). By analyzing the dead and live variables, global states which have the same values for live variables and different values for dead variables can be treated as the same.

Based on the CFSM model, Huang *et al.* have proposed an incremental protocol verification method that shows its effectiveness when the modification is small (Huang *et al.*, 1990). Using the CFSM-based incremental protocol verification method, an incremental protocol design environment has been developed and executed on the OPS5 production system (Frogy, 1982; Gupta *et al.*, 1988; Acharya *et al.*, 1992). Incremental protocol verification means that protocol verification is continued incrementally at the modification point. In the traditional non-incremental verification approach, whenever one or more transitions have been modified, the global state reachability analysis should be re-started from the initial global state. All of the global states, including those states that are error-free and those global states that are not related to the modification, should be re-explored. Even if there are $1000*n$ or $10000*n$ global states and only one global state contains an error, all of the $1000*n$ or $10000*n$ global states need to be re-explored once again after the modification. Incremental verification can process those global states that are related to the modification and continue the global state reachability analysis incrementally at the modification point.

Since the current Formal Description Techniques (FDTs) (Sidhu, 1989a; Chanson *et al.*, 1993), such as

ISO's Estelle (Budkowski and Dembinski, 1987; ISO, 1987; Linn, 1988; Algayers *et al.* 1993) and CCITT's SDL (Belina and Hogrede, 1984; CCITT, 1988), are based on the extended state transition model, e.g. the Extended Communicating Finite State Machine (ECFSM) model, the incremental protocol verification method must be modified in order to be applied to the ECFSM-based FDTs. By integrating the concept of dead and live variables in Chu and Liu's global state reduction technique (Chu and Liu, 1989), a new ECFSM-based incremental protocol verification method is proposed in this paper. Using the dead and live variables concept, global states that have the same values for live variables and different values for dead variables can be treated as the same. There are two main concerns in the proposed ECFSM-based incremental protocol verification method that incorporates the dead and live variables concept: (1) the change of equivalent global states' equivalence that results from some dead (live) variables becoming live (dead) ones and (2) the effect on the global state reachability graph that results from adding/deleting transitions. For the first concern, some originally equivalent global states may become unequivalent due to some dead variables becoming live ones and some originally unequivalent global states may become equivalent due to some live variables becoming dead ones. For the second concern, some global states should be deleted on the deleting transitions case, some unextendable global states may become extendable on the adding transitions case and some logical errors can be removed, but new logical errors may be generated on both adding and deleting transitions cases.

Based on the new method, incremental protocol verification can be directly applied to the ECFSM-based FDTs, i.e. Estelle and SDL, to have FDT-based incremental protocol design environments. Using the proposed new ECFSM-based incremental protocol verification method, we have developed an Estelle-based incremental protocol design system on SUN SPARC workstations. The rest of the paper is organized as follows. Section 2 gives the formal model of the ECFSM and a brief review of Chu and Liu's global state reduction technique. Section 3 presents the new ECFSM-based incremental protocol verification method. Section 4 briefly introduces the application of the new method to Estelle and gives an example to show the usage. Section 5 discusses issues of the new incremental protocol verification method. Finally, Section 6 has the conclusion remarks.

2. PRELIMINARY

In this section, the formal model of the ECFSM and the related definitions are introduced at first. Then, a brief review of Chu and Liu's global state reduction technique using the dead and live variables concept is presented.

2.1. ECFSM

In the ECFSM model, the behavior of each protocol entity is described as an ECFSM with a set of context variables that can be accessed during state transition. Each state transition in the ECFSM-based model is associated with a head state, a tail state and a label consisting of two parts: a condition part followed by an action part. The condition part consists of an input event and/or a predicate. The action part consists of a sequence of statements that operate on context variables and/or output events. A transition in the ECFSM model is executable when the entity has reached the head state of a transition and the condition part of the transition becomes TRUE. A condition becomes TRUE if and only if its component predicate and input event become TRUE. The context variables of each ECFSM are independent and there are no globally shared variables among the ECFSMs. Protocol entities communicate with each other by message passing through a number of First-In-First-Out (FIFO) unidirectional queues (channels). Figure 1 shows an Alternating Bit Protocol (ABP) that is formally specified in the ECFSM model, where '?' represents input, '!' represents output, a circle represents a state, and an arrow represents a transition.

Each ECFSM can be represented as a seven-tuple $(\Sigma, S, s_0, V, P, A, \delta)$, where

1. Σ is the set of messages that can be sent or received.
2. S is the set of states.
3. s_0 is the initial state of the ECFSM.
4. V is the set of context variables.
5. P is the set of Boolean expressions that operate on context variables.
6. A is the set of actions that operate on context variables,
7. δ is the set of transition functions, where each transition function is generically represented as follows: $S \times \Sigma \times P(V) \rightarrow \Sigma \times A(V) \times S$.

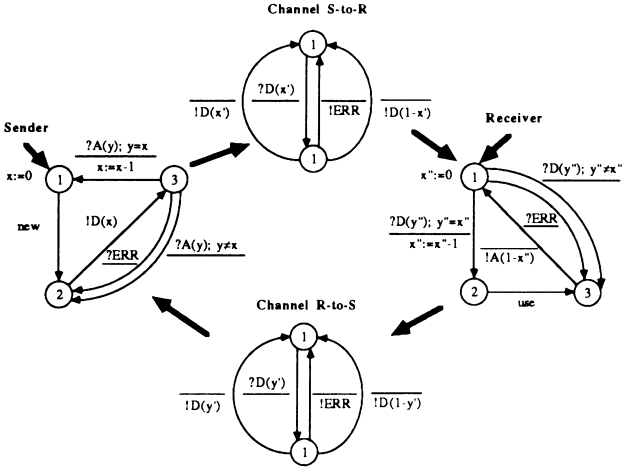


FIGURE 1. An ABP that is formally specified in the ECFSM model.

The transitions in the ECFSM model can be classified into two types:

- Spontaneous transition: the transition whose condition part has no input event, i.e. $S \times P(V) \rightarrow \Sigma \times A(V) \times S$.
- When transition: the transition whose condition part has an input event with/without a predicate, i.e. $S \times \Sigma \times P(V) \rightarrow \Sigma \times A(V) \times S$.

Consequently, the states in the ECFSM-model can be classified into three types:

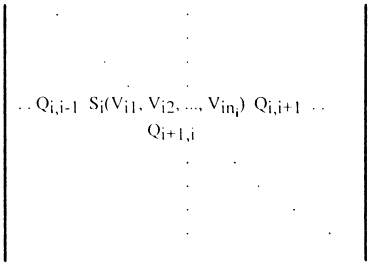
- Spontaneous state: the state whose outgoing transitions are all spontaneous transitions.
- When state: the state whose outgoing transitions are all when transitions.
- Mixed state: the state whose outgoing transitions consist of spontaneous and when transitions.

In the ECFSM model, a global state is represented as an $n \times n$ matrix, where n is the number of entities. Each element in the diagonal records the state of the corresponding entity and the values of the context variables declared in the entity; each element in the off-diagonal records the sequence of messages with appended parameter values in transit on the corresponding queue. Figure 2 shows the abstract format of a global state structure in the ECFSM model. For convenience, let $GS_I - e \rightarrow GS_J$ represent transition e is executed at global state GS_I and global state GS_J is generated after the execution of e, where GS_I is called the parent state and GS_J is called the child state, e is called an outgoing transition of GS_I and e is also called an incoming transition of GS_J.

2.2. Review of Chu and Liu's reduction technique

In the ECFSM model, a context variable is accessed in two ways: reference and assignment. A context variable is *referenced* (or read) when it is accessed in a predicate, in the right-hand side of an infix assignment operator, e.g. ':=' operator in Pascal, or in the parameter list of a send event. A context variable is *assigned* (or written) when it is accessed in the left-hand side of an infix assignment operator or in the parameter list of a receive event.

Let x be a context variable of entity a and p be a state of the ECFSM describing entity a. Then x is dead at p if, starting from state p, x will not be referenced in all possible future execution paths of entity a or the next possible accesses to x in all possible future execution paths of entity a are all assignments. For example, in the Alternating Bit Protocol (ABP) that is depicted in Figure 1, variable y is dead at every state of entity *Sender*. This is because y is referenced in two transitions, one is from state 3 to state 1 and the other one is from state 3 to state 2; in the labels of both transitions the reference to y is immediately preceded by a receive event assigning a value to y. Hence, all possible next accesses to y are



S_i: the state of entity i, i = 1, ..., N.
n_i: the number of context variables declared for entity i, i = 1, ..., n.
V_{ik}: the k_{th} context variable of entity i, i = 1, ..., n, k = 1, ..., n_i.
Q_{ij}: communication channel from entity i to entity j, i, j = 1, ..., n, i ≠ j.

FIGURE 2. The structure of a global state matrix.

assignments starting from every state of entity *Sender*. By the same argument, y'' is dead at every state of entity *Receiver*.

It is observed that the future behavior of entity a, starting from p, does not depend on the current value taken on by a variable x that is dead at p. The reason is that the other variables that are not dead will not be influenced or changed their taken values by the dead variables. Therefore, any two global states whose corresponding element values are identical to each other except the values taken on by the dead context variables can be considered as equivalent (Chu and Liu, 1989).

3. INCREMENTAL PROTOCOL VERIFICATION FOR THE ECFSM MODEL

In order to apply the incremental processing for the ECFSM model, context variables and predicates should be taken into consideration in the incremental verification method. Furthermore, in order to have more efficient incremental processing for the ECFSM model, the incremental verification method is augmented with the dead and live variables concept. In this section, logical errors that can be detected are briefly introduced at first. Then, the new ECFSM-based incremental protocol verification method is presented.

3.1. Logical errors

In our proposed ECFSM-based global state reachability analysis, some logical errors can be detected. These logical errors are defined as follows (assume n is the number of protocol entities):

A global state contains a *deadlock error* if the following conditions are satisfied:

1. All of the communicating channels $Q_{I \rightarrow J}$, where $I = 1, \ldots, n$, $J = 1, \ldots, n$, $I \neq J$, are empty.
2. All of the current state S_I of entity I, $I = 1, \ldots, n$, have no outgoing spontaneous transition, or some of them have outgoing spontaneous transitions but the associated predicates are false.
3. There is a state S_I of entity E_I such that S_I is not a terminal state, when terminal states are defined in the protocol.

A global state contains an *unspecified reception error* if the following conditions are satisfied.

1. If there is an I, such that all of the head messages of the communicating channel $Q_{J\to I}$, $J = 1,\ldots,n$, $J \neq I$, are unspecified in the current state S_I of entity E_I, or some of the head messages of communicating channel $Q_{J\to I}$, $J = 1,\ldots,n$, $J \neq I$, are specified but their associated predicates are false.
2. The current state S_I of entity E_I has no outgoing spontaneous transition, or has some outgoing spontaneous transitions but their associated predicates are false.

A global state contains a *transmitted lock error* if the following conditions are satisfied:
For an entity I, $I = 1,\ldots,n$,

1. All of the outgoing transitions are spontaneous transitions.
2. All of the associated predicates are false.

A global state contains a *channel overflow error* if the number of messages in a communicating channel is greater than the channel size.

3.2. The incremental protocol verification algorithm

Incremental protocol verification is divided into two parts: adding new transitions and deleting old transitions. Adding new transitions can eliminate some logical errors. For simplicity, the partial global state graph shown in Figure 3 is used for explanation. For example, adding a spontaneous transition can make a global state, which contains a head state of an added transition, with a deadlock error to be error-free. In Figure 3, assume global state R_1 contains a deadlock error. Then, if a transition W that is executable in R_1 is added, then R_1 becomes extendable and R_1 is not an erroneous state any more. Additionally, adding new transitions may also generate new global states from the global states that contain the head states of the added transitions. Therefore, adding new transitions may have a side effect of generating new errors.

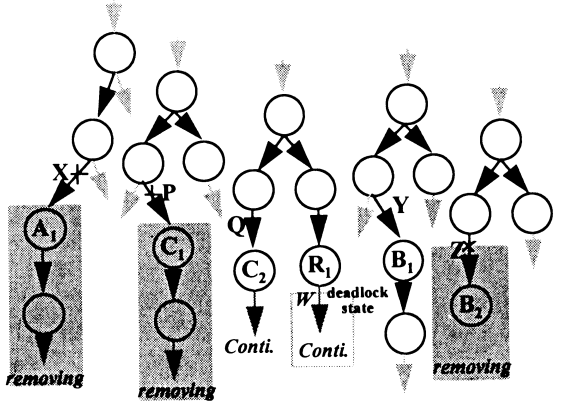


FIGURE 3. The partial global state graph used for explaining adding/deleting transitions.

Deleting transitions can eliminate logical errors too. Deleting transitions can remove those global states whose incoming transitions belong to the deleted transitions. However, deleting transitions may also have a side effect of generating new logical errors. For example, if a global state's communicating queues are all empty and all of its executable outgoing transitions belong to the deleted transitions, then the global state may have a deadlock error after the modification. Therefore, the parent states of the deleted states need to be checked. Moreover, all of the global state sequences rooted from the deleted global states need to be removed in the global state graph. The global states that should be removed when a transition is deleted can be classified into three classes. For simplicity, the partial global state graph shown in Figure 3 is still used for explanation.

1. *The removed global state is unique in the global state graph.* In this case, this global state should be removed and all of the global state sequences rooted from this state should be removed too. For example, if transition X is deleted, then global state A_1, which is generated by executing X and is unique in the global state graph, and those global state sequences rooted from A_1 should be removed.
2. *The removed global state is the first occurrence of a set of equivalent global states.* In this case, this global state and all of the global state sequences rooted from this state are removed, and one of the equivalent states, i.e., a second occurrence that is not generated by executing the deleted transitions, is promoted as the first occurrence. For example, if transition P is deleted, global state C_1 and those global state sequences rooted from C_1 are removed. Since C_1 has an equivalent state C_2, which is generated by executing another transition Q, C_2 can be promoted as the first occurrence.
3. *The removed transition is the second occurrence of a set of equivalent global states.* In this case, this global state is removed without any side effect. For example, if transition Z is deleted, then global state B_2 should be removed. Global state B_1, which is the same as B_2 and is the first occurrence and is generated by executing transition Y, is not affected.

Adding/deleting transitions may result in the change of dead and live variables sets. As a result, some originally equivalent global states may become unequivalent due to their corresponding dead variables becoming live ones; some originally unequivalent global states may become equivalent due to their corresponding live variables becoming dead ones. For convenience, the partial global state graph depicted in Figure 4 is used for explanation.

1. *Dead variables becoming live ones.* Assume variable y is dead originally. Using the dead and live variables concept, global states GS_A and GS_B are equivalent. Let variable y become live after modification.

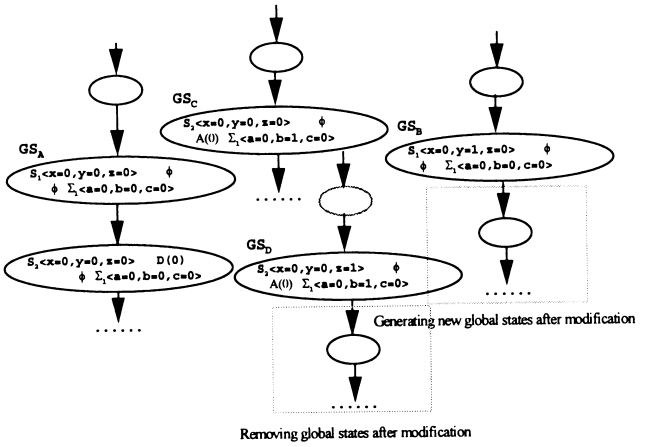


FIGURE 4. The partial global graph used for explaining dead and live variables change.

Consequently, GS_B and GS_A become unequivalent, because y's values are different in GS_A and GS_B. Thus, GS_B should be explored, i.e., GS_B becomes extendable, after modification.

2. *Live variables becoming dead ones.* GS_C and GS_D are originally unequivalent. Let variable z become dead after modification. Consequently, GS_C and GS_D become equivalent because their element values are equal except the value taken on by the dead variable z. Assume GS_D becomes the second occurrence. Thus, GS_D's descendant state sequences are removed because GS_D is the second occurrence.

For convenience, some notations are used to explain the ECFSM-based incremental protocol verification algorithm. EXTENDABLE is a pool storing those global states that have not been explored. GLOBALSTATE is a pool storing those global states that have already been explored.

Algorithm **G-S-R-A**, which is shown below, formalizes the global state reachability analysis with the concept of dead and live variables for n-entity protocols. There are three main steps in the global state reachability analysis. Step 1 evaluates dead and live variables of each state of the n ECFSMs. Step 2 initializes the initial global state. Step 3 is the main step of G-S-R-A. Each global state in Extendable is checked. If a global state GS has no executable outgoing transition, then GS's logical correctness is checked in Step 3. Depending on designer's convenience, incremental verification processing can be invoked if logical errors are detected in Step 3-1. If a global state GS is extendable, all of GS's executable transitions are executed in Step 3-2. The equivalent property of each newly generated child global state of GS is checked based on the dead and live variables concept.

Algorithm G-S-R-A:

Step 1: Evaluate dead and live variables of each state of $ECFSM_I$, $I = 1, \ldots, n$.

Step 2: Initialize the initial global state GS_{init} and add GS_{init} to EXTENDABLE.

Step 3: **while** EXTENDABLE is not empty **do**
  Remove a global state GS from EXTENDABLE.
  if GS does not have any executable transition **then**

Step 3-1: **case of**
  GS is not a terminal state:
    Mark GS as an erroneous state according to the error type and add GS to GLOBALSTATE.
    if the designer wants to modify the protocol at this point **then**
      Call the *incremental verification processing*.
  GS is a terminal state:
    Mark GS as a terminal state and add GS to GLOBALSTATE.
  endcase
else

Step 3-2: **for** each executable outgoing transition e of GS such that $GS - e \rightarrow GS'$ do
  if there is a GS_I in EXTENDABLE or GLOBALSTATE that is marked as unique or the first occurrence such that GS_I is equal to GS' except for the values taken on by dead variables
  then
    Mark GS_I as the first occurrence if it is originally unique.
    Mark GS' as the equivalent state of GS_I and add GS' to GLOBALSTATE.
  else
    Mark GS' as an unique state and add GS' to EXTENDABLE.
  endif
  endfor
endif
endwhile

Incremental verification processing contains seven steps, which are described as follows:

Step 1: Modify the protocol: add transitions and/or delete transitions.
Step 2: Evaluate dead and live variables of each state of the modified $ECFSM_I$, $I = 1, \ldots, n$.
Step 3: Analyze the effect that is resulted from some dead variables becoming live ones.
Step 4: Analyze the effect that is resulted from some live variables becoming dead ones.
Step 5: Inspect the logical correctness of the related global states and modify the associated information, which are resulted from deleting transitions.

Step 6: Inspect the logical correctness of the related global states and modify the associated information, which are resulted from adding transitions.

Step 7: Go to Step 3-1 of algorithm **G-S-R-A**.

Step 1 adds and deletes transitions. Step 2 reevaluates the dead and live variables of each state after adding and deleting transitions. Step 3 calculates those global states that are originally regarded as equivalent but become unequivalent due to their dead variables becoming live ones. The details of Step 3 are presented in the algorithm **Dead-to-Live-Analysis**, which is shown below. A global state GS that has the condition of dead variables becoming live ones may become equivalent to other global states after the dead-to-live change. Step 1 deals with this situation. If GS becomes other global state's equivalent state, GS's child states need to be removed because GS becomes the second occurrence. Procedure **Clear** removes the descendant global state sequences rooted from the global states that have to be removed. If GS becomes other global state's equivalent state, GS's originally equivalent states need to be checked. That is, those states which are originally regarded as equivalent to GS may be regarded as unequivalent due to the corresponding dead variables becoming live ones. Step 2 deals with the condition. In Step 2-1, if an originally equivalent state GS'' becomes unequivalent to GS due to the dead-to-live effect, GS'' is checked to see whether it becomes an equivalence of another state. In Step 2-2, if GS'' is still equivalent to GS, then GS'' becomes an equivalent state of GS' in case GS becomes an equivalent state of GS' due to the dead to live effect.

Algorithm Dead-to-Live Analysis:

for each GS that is marked as unique or the first occurrence in EXTENDABLE or GLOBALSTATE and GS's component state contain a state S such that S has dead variables becoming live ones **do**

Step 1: **if** there is a GS' in GLOBALSTATE or EXTENDABLE which is marked as unique or the first occurrence such that GS is equal to GS' except the values taken on by the dead variables and GS' is not a descendant state of GS
then
Mark GS' as the first occurrence if GS' is originally unique.
Mark GS as an equivalent state of GS'.
for each child (GS) **do**
Clear(child(GS), all deleted transitions)
endfor
endif

Step 2: **for** each state GS'' that is an equivalent state of GS **do**
if GS'' becomes different from GS due to the dead-to-live change

Step 2-1: **then**
if there is a GS''' in GLOBALSTATE or EXTENDABLE that is marked as unique or the first occurrence such that GS''' is equal to GS'' except the values taken on by the dead variables
then Mark GS'' as the equivalent state of GS'''.
else Add GS'' to EXTENDABLE.

Step 2-2: **else**
if GS becomes an equivalent state of GS' **then**
Mark GS'' as an equivalent state of GS'
endif
endif
endfor
endfor

Procedure Clear(GS, T):

Step 1: Remove GS from GLOBALSTATE or EXTENDABLE.

Step 2:
case of GS
A unique state:
for each child(GS) **do**
Clear(child(GS), T).
endfor
The first occurrence:
Promote one of the equivalent state GS' of GS that is not GS's descendant state and is not generated by executing transition T as the first occurrence;
Add GS' to EXTENDABLE.
for each child(GS) **do**
Clear(child(Gs), T).
endfor
The second occurrence:
Do nothing.
endcase

Step 4 of the incremental verification processing calculates those global states that become equivalent due to their live variables becoming dead ones. The details of Step 4 is presented in algorithm **Live-to-Dead-Analysis**, which is shown below. For those global states that have been checked in Step 3, they do not need to be checked again. That is, in case a global state contains some variables whose attributes changed from dead to live and some variables whose attributes changed from live to dead, this global state is checked in Step 3. After the selection, each of the affected global states GS is checked to see whether GS becomes other states' equivalent state or not. If the answer is positive, then the equivalence information of GS's originally equivalent states are modified, and GS's descendant state sequences are removed.

Algorithm Live-to-Dead-Analysis:

for each GS that is marked as unique or the first occurrence in EXTENDABLE or GLOBALSTATE

and GS's component states don't have dead variables becoming live ones but contain a state S such that S has live variables becoming dead ones **do**
if there is a GS' in EXTENDABLE or GLOBALSTATE that is marked as unique or the first occurrence such that GS' is equal to GS except the values taken on by dead variables and GS' is not a descendant state of GS
then
Mark GS' as the first occurrence if it is originally unique.
Mark GS as an equivalent state of GS'.
Modify each equivalent state of GS as an equivalent state of GS'.
for each child(GS) **do**
Clear(child(GS), all deleted transitions).
endfor
endif
endfor

Step 5 of the incremental verification processing inspects the logical correctness of the related global states and deletes those states that need to be removed for the deleting transitions case. The details of Step 5 is presented in Algorithm **Delete-Process**, which is shown below. As mentioned previously, logical errors can be eliminated when the incoming transitions of the corresponding global states are deleted. But, some new errors may also be generated after deleting transitions. For example, let global state GS be generated by executing transition e, which has been deleted, from global state GS_p. If GS_p has only one executable outgoing transition, i.e. the deleted transition e, and GS_p is not a terminal state, then GS_p becomes an erroneous state. Step 1 has the logical correctness inspection of the related global states. After the inspection, Step 2 removes those global states, which are generated by executing the deleted transitions, and their descendant states.

Algorithm Delete-Process:
for each deleted transition e **do**
for each GS in GLOBALSTATE or EXTENDABLE such that $GS_p - e \rightarrow GS$ **do**
Step 1: **if** GS_p becomes an erroneous state
then
Mark GS_p as an erroneous state according to the error type
endif
Step 2: Remove GS from GLOBALSTATE or EXTENDABLE and delete the associated errors.
for each child(GS) **do**
Clear(child(GS), all deleted transitions).
endfor
endfor
enfor

Step 6 of the incremental processing selects those global states that become re-extendable for the adding transitions case. Algorithm **Add-Process**, which is shown below, depicts the details of Step 6. When an entity's state in a global state GS is the head state of an added transition, GS becomes re-extendable even if GS has originally been explored.

Algorithm Add-Process:
for each global state GS that is unique or first occurrence in GLOBALSTATE **do**
if GS's component states contain a state S that is equal to the head state of a new transition T
then
Delete the corresponding errors associated with GS that will not occur after the new transition being added.
Add GS to EXTENDABLE.
endif
endfor

After selecting the re-extendable global states, Step 7 of incremental verification processing returns to Step 3-1 of Algorithm **G-S-R-A** to explore new global states and detect additional logical errors.

4. APPLICATION AND USAGE

In this section, we briefly present the application of the new incremental verification method to ISO's Estelle at first. Then, an example is given to show the usage.

4.1. Application

Based on the new incremental protocol verification method proposed in this paper, we have developed an incremental Estelle translator on SUN SPARC workstations (Huang *et al.*, 1993). In this way, an Estelle-based incremental protocol design system is achieved.

The framework of an Estelle specification is a set of cooperating entities. Figure 5 shows the abstract format of an Estelle specification. Each entity is described as a module. Each module is attributed with *systemprocess*, *systemactivity*, *process* or *activity*. Entitites interact with each other by exchanging messages through channels. A channel is full-duplex that transmits messages between two connected modules. The actual behavior of a module is specified as a process. An Estelle process definition specifies a queue discipline associated with each interaction point, the initial condition and all possible transitions for the corresponding ECFSM. The general transitions form of Estelle is also depicted in Figure 5, in which TRANS is a keyword delimiting the transition declaration part, FROM is the current state, TO is the next state, WHEN is the input event, PROVIDED is the predicate part to be satisfied and ACTION is the action part to be executed.

In order to have an efficient and automatic verification system, a subset of Estelle is supported in our system. Based on the similar concerns presented in (Courtiat, 1987, 1988; Courtiat and Saqui-Sannes, 1992; Algayers

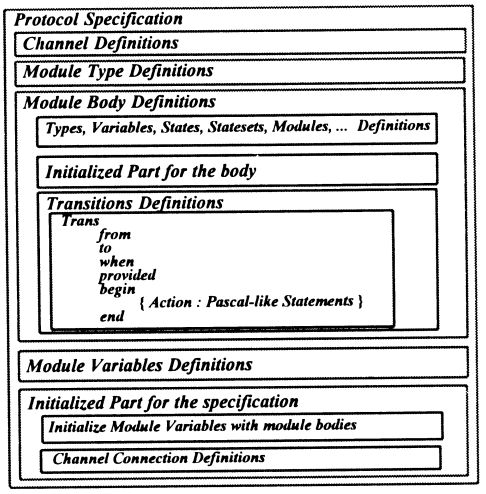


FIGURE 5. The abstract format of an Estelle specification.

et al., 1993), the Estelle supported in our system has the following restrictions:

1. As it is mentioned in (Couriat, 1987, 1988; Courtiat and Saqui-Sannes, 1992), global synchronization for *systemprocess* modules may lead to undesired over-specifications. Hence, *systemprocess* and *process* modules are not supported. Only *systemactivity* and *activity* attributes are supported to restrict the execution to have interleaving semantics (Algayers *et al.*, 1993).
2. Dynamic features are not allowed, i.e. module instances cannot be created dynamically. In other words, a static configuration which is similar to that in (Algayers *et al.*, 1993) is supported.
3. In order to have fully automatic execution, incomplete definitions of functions/procedures are not allowed, and 'any' and '...' undefined types are not allowed either.

The Estelle translator interprets the Estelle specification to be executable, analyzes the dead and live variables of each state of each entity and so on. The Estelle translator contains the following functionalities:

1. *An ECFSM table generator*. It can generate the ECFSM table in which ECFSMs are linked as a link list and therefore the number of entities can be decided from the table. Each ECFSM is also built as a tree. The ECFSM table contains the module bodies and interaction points declared in the Estelle specification.
2. *A module body generator*. It can generate the module bodies declared in the Estelle specifications. Each module body is a structure that contains some data and pointer fields. The module body contains the initial state, state list, variable list, and transition list.
3. *A communication channel generator*. It can generate the channels between entities from the channel declaration part of the Estelle specifications. The channel structure is important to decide the communicating role, e.g. subject or object entity. Each channel is uni-directional and First-In-First-Out (FIFO).
4. *A dead and live variables generator*. It can decide dead and live variables for each state. The process of seeking dead and live variables is divided into two stages. In the first stage, all transitions are traversed to decide each variable's first access type, i.e. assigned or referenced, for each transition. In the second stage, each ECFSM graph is traced to decide the dead variables of each state of each entity.
5. *A global state structure generator*. It can decide the number of queues from the communication channel generator and the number of ECFSMs from the ECFSM table. The communication queues of the entities that have been connected using a 'CONNECT' statement should exist in the global state structure, otherwise they can be eliminated.

Based on the Estelle translator, we have developed an associated protocol design system on SUN SPARC workstations. A Graphical-User-Interface (GUI) is also provided. The main features of the associated protocol design system are as follows:

1. Protocol designers can load Estelle specifications in text files or interactively edit Estelle-specified protocols using the provided editor.
2. Protocol designers can (i) control, e.g. set the number of generated (erroneous) global states as the temporary halt point, (ii) observe, e.g. display a list of generated (erroneous) global states, or a list of specified transitions and (iii) trace, e.g. display the transition sequences from the initial global state to an erroneous global state, the execution of protocol verification (Figures 6 and 7).
3. Protocol designers can perform adding/deleting transitions at the temporary halt point. That is,

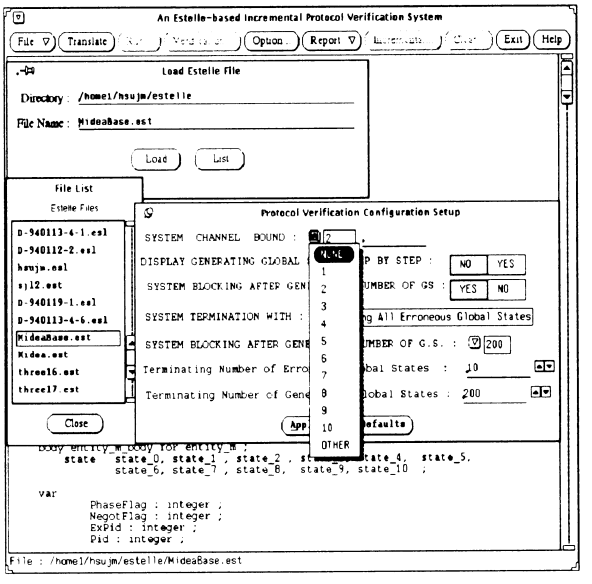


FIGURE 6. Setting some control parameters.

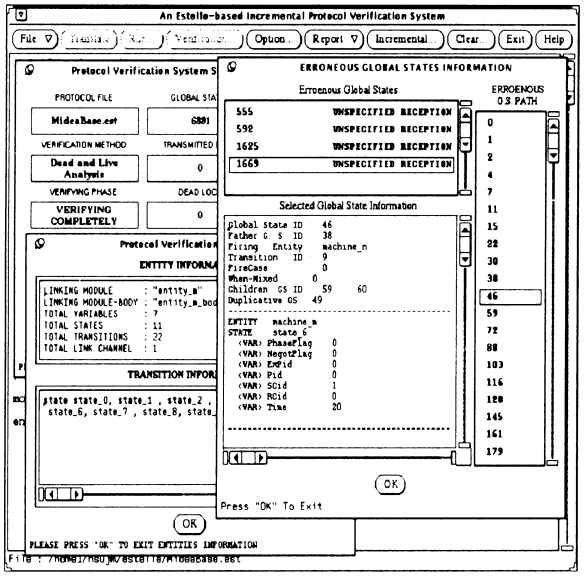


FIGURE 7. Observing and tracing verification result.

protocol designers can edit the added transitions that are specified in Estelle using the provided GUI. Protocol designers can also use the GUI to list all of currently existing transitions and delete the corresponding transitions for removing the associated logical errors (Figure 8).

The incremental verification process is invoked optionally when logical errors are detected. The incremental verification process compares the new dead and live variables with the old dead and live variables, identify the equivalent properties among global states, delete some global states and the associated logical errors, identify those global states that become re-extendable and so on.

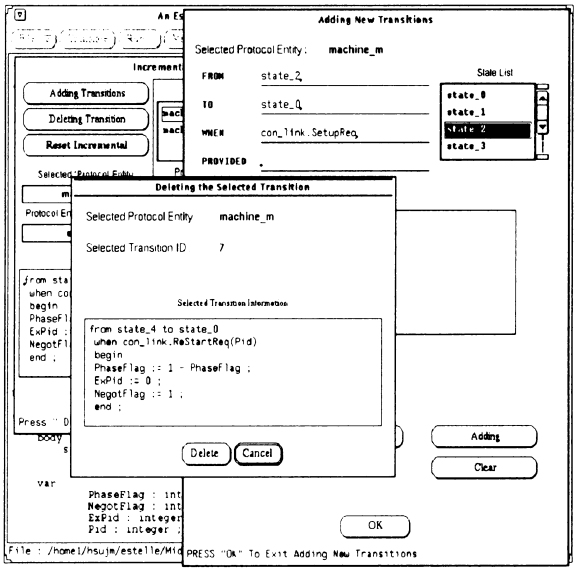
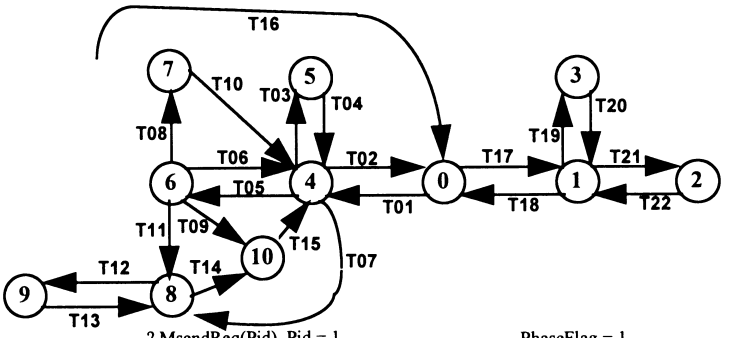


FIGURE 8. Adding and/or deleting transitions.

4.2. Usage

Figure 9 shows a protocol used for the multimedia application, in which the client (user) side sends text-based data to the server (provider) side, the server (provider) side tries to send image-based or video-based data to the client (user) side periodically. The underlying network is an ethernet-based bus. There are two users, i.e. User 0 and User 1, on the client side. Each one holds bus usage alternatively. There are also two providers, i.e. Server 0 and Server 1, on the server side. These two providers try to send data for every T_1 time units (delay (T_1) in transition $S05$) alternatively. Since the data sent from server to client, i.e. image-based or video-based data, always need a long time period, a provider is allowed to send when its paired users, which is coupled by the id (0 or 1), has hold the bus usage once, or when both users in the client side don't want to use the bus. Since the amount of data from server to client is very large, the server side should send a request message to users to hold the bus usage. After receiving the acknowledgement, a provider can prepare to send. The ideal scheme is pure interleaving: $(user0 \overset{then}{\rightarrow} server0$ or $server0 \overset{then}{\rightarrow} user0) \overset{then}{\rightarrow} (user1 \overset{then}{\rightarrow} server1$ or $server1 \overset{then}{\rightarrow} user1)$, and the cycle is repeatedly executed. However, the client side has the higher privilege to use the bus according to users' need. Since both sides can initiate the usage as they like, some collisions situations may occur when both sides try to hold the bus at the same time. When the number of collisions is greater than N, the server side will initiate a negotiation, e.g. renew the receive buffer size, transmission speed, and etc. The following usage scheme is used in the protocol to resolve the bus usage: where $(trans_1, trans_2)$ represents transitions $trans_1$ and $trans_2$ colliding; function *GetCounter( ) in transitions S07 and S12 decides how many data transmission sessions the server are allowed in the current cycle;* $(trans_1 \overset{then}{\rightarrow} \cdots \overset{then}{\rightarrow} trans_n)^n$ *represents transition sequence* $trans_1 \overset{then}{\rightarrow} \cdots \overset{then}{\rightarrow} trans_n$ *is repeatedly executed for n times.*

- Pure interleaving: As mentioned above.
- If Server 0 requests, but User 0 does not want to use, then Server 1 requests. In this case, Server 0 can start to send. After User 0 sending a respond message and switching to User 1, then Server 1 can start to play. The transition sequence is $S05 \overset{then}{\rightarrow} T03 \overset{then}{\rightarrow} T04 \overset{then}{\rightarrow} S06 \overset{then}{\rightarrow} S05 \overset{then}{\rightarrow} T03 \overset{then}{\rightarrow} T04 \overset{then}{\rightarrow} S07 \overset{then}{\rightarrow} T07 \overset{then}{\rightarrow} (S13 \overset{then}{\rightarrow} T12 \overset{then}{\rightarrow} T13 \overset{then}{\rightarrow} S15)^n \overset{then}{\rightarrow} S14 \overset{then}{\rightarrow} T14 \overset{then}{\rightarrow} T15 \overset{then}{\rightarrow} S11 \overset{then}{\rightarrow} T05 \overset{then}{\rightarrow} S03 \overset{then}{\rightarrow} S04 \overset{then}{\rightarrow} T06 \overset{then}{\rightarrow} \ldots$, where n is decided by *GetCounter( )* in the transition S_{07}. Otherwise, i.e. User 0 does not send any message, Server 1 cannot play and lose the chance after the predefined time interval, i.e. T_1.
- If User 0's sending data collides with Server 0's request message, User 0 gets the usage, Server 0 needs to retry again. The transition sequence is $(S05, T05) \overset{then}{\rightarrow} T08 \overset{then}{\rightarrow} T10 \overset{then}{\rightarrow} S08 \overset{then}{\rightarrow} S09$.
- If User 1's sending data collides with Server 0's



Client Site :
all variables' initial values are 0
initial state : 0

T01 : delay(T_3) , PhaseFlag = 0 / ! StartReq(SCid)

T02 : ? ReStartReq (Pid) / PhaseFlag:=1 - PhaseFlag ; ExPid:=0 NegotFlag:=1

T03 : ? MsendReq(Pid) / ExPid := 1 - ExPid

T04 : / ! MsendAck (1 - ExPid)

T05 : / ! Tdata(SCid)

T06 : ?TdataAck(RCid) / SCid := 1 - SCid

T07 : ? ReadytoSend (Pid) /

T08 : ? MsendReq(Pid) , Pid = 0 / SCid := 1 - SCid

T09: ? MsendReq(Pid), Pid = 1 / ExPid:=1-ExPid; SCid := 1 - SCid !MsendAck (1-ExPid)

T10 : / ExPid:= 0 ; !Collide (SCid)

T11 : ? ReadytoSend (Pid). / SCid := 1 - SCid

T12 : ? Mdata /

T13 : / ! MdataAck

T14 : ? Resmue /

T15 : / ! NextCycle

T16 : ? ReStartReq (Pid) / PhaseFlag:=1-PhaseFlag; ExPid := 0 NegotFlag:=1 ; SCid := 1 - SCid

T17: PhaseFlag = 1 / ! NegotReq

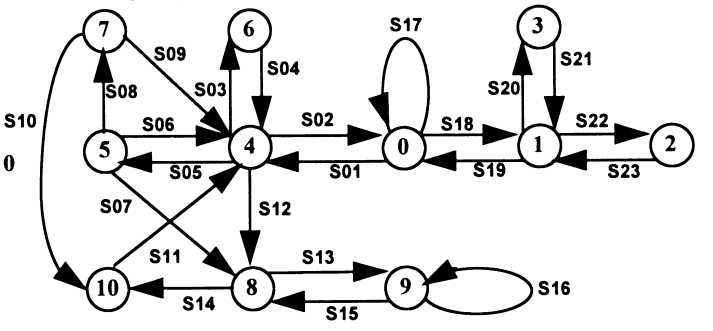
T18: ? NegotOver / NegotFlag := 0

T19 : ? SetupReq, PhaseFlag = 1 /

T20 : / !SetupAck; PhaseFlag := 1 - PhaseFlag

T21 : PhaseFlag = 1 / !SetupReq

T22 : ? SetupAck / PhaseFlag := 1 - PhaseFlag



Server Site :
variable *ReadyFlag*'s intial value is 1
all of the other variables' initial values are 0
initial state : 0

S01 : ? StartReq(Cid) / NeogtFlag := 0 PhaseFlag := 1 - PhaseFlag

S02 : NegotFlag =1 / !RestartReq(SPid) ; SPid := 0 ReadyFlag:=1

S03 : ? Tdata (Cid) / ExCid := 1 - ExCid ;

S04 : / ! TdataAck (1-ExCid)

S05 : delay(T_1) , NegotFlag = 0 & ReadyFlag = 1 / ! MsendReq (SPid) ; ReadyFlag := 0

S06 : ? MsendAck(RPid) , RPid = 0 / SPid := 1 - SPid ;

S07 : ? MsendAck(RPid) , RPid = 1 / SPid := 1 - SPid;Counter := GetCounter() !ReadytoSend(SPid)

S08 : ? Tdata (Cid) / SPid := 0 ; TryCounter := TryCounter + 1 IF TryCounter > *N* then *begin* TryCounter := 0 ; NegotFlag := 1 *end*

S09: ? Collide(Cid) / SPid := 0; ReadyFlag := 1

S10 : ? MsendAck(RPid) /

S11 : ? NextCycle / ReadyFlag := 1

S12 : SPid = ExCid & ReadyFlag := 0 / ! ReadytoSend(SPid) Counter:=GetCounter()

S13 : Counter > 0 / !Mdata ; Counter := Counter -1

S14 : Counter = 0 / ! Resume

S15 : ? MdataAck /

S16 : ? Tdata (Cid) / ExCid := 1 - ExCid

S17: ? Tdata(Cid) / ExCid := 1 - ExCid

S18: ? NegotReq , PhaseFlag = 1 /

S19 : PhaseFlag = 0 / !NegotOver

S20 : ? SetupReq /

S21 : / PhaseFlag := 1 - PhaseFlag !SetupAck

S22 : delay(T_2) , PhaseFlag = 1 / ! SetupReq

S23 : ? SetupAck / PhaseFlag := 1 - PhaseFlag ;

FIGURE 9. The protocol used for explanation.

request message after User 0 has used the bus, Server 0 needs to retry again. The transition sequence is $T05 \xrightarrow{then} S03 \xrightarrow{then} S04 \xrightarrow{then} T06 \xrightarrow{then} (S05, T05) \xrightarrow{then} T08 \xrightarrow{then} T10 \xrightarrow{then} S08 \xrightarrow{then} S09$.

- If Server 0 requests, then User 0's sending data collides with Server 1's request message. In this case, User 0 can send data and Server 0 needs to retry again. The transition sequence is $S05 \xrightarrow{then} T03 \xrightarrow{then} T04 \xrightarrow{then} S06 \xrightarrow{then} (S05, T05) \xrightarrow{then} T09 \xrightarrow{then} T15 \xrightarrow{then} S08 \xrightarrow{then} S10 \xrightarrow{then} S11$.
- If Server 0 (Server 1) starts to send, then User 1 (User 0) wants to send, then User 1 (User 0) is granted (the period is short comparing with that of sending image or video data, so let users to use the bus), but User 0 (User 1) can send after Server 0's (Server 1's) sending data has finished. The transition sequence is $\ldots \xrightarrow{then} (S12, T05) \xrightarrow{then} S13 \xrightarrow{then} T11 \xrightarrow{then} T12 \xrightarrow{then} T13 \xrightarrow{then} S16 \xrightarrow{then} S15 \xrightarrow{then} (S13 \xrightarrow{then} T12 \xrightarrow{then} T13 \xrightarrow{then} S15)^n \xrightarrow{then} \ldots$.
- If Server 0 requests, User 0 sends data, Server 0 sends data, then User 1's sending data collides with Server 1's requests message. In this case, Server 1 needs to try again. The transition sequence is $S05 \xrightarrow{then} T03 \xrightarrow{then} T04 \xrightarrow{then} S06 \xrightarrow{then} T05 \xrightarrow{then} S03 \xrightarrow{then} S04 \xrightarrow{then} T06 \xrightarrow{then} S12 \xrightarrow{then} T07 \xrightarrow{then} (S13 \xrightarrow{then} T12 \xrightarrow{then} T13 \xrightarrow{then} S15)^n \xrightarrow{then} S14 \xrightarrow{then} T14 \xrightarrow{then} T15 \xrightarrow{then} S11 \xrightarrow{then} (S05, T05) \xrightarrow{then} T09 \xrightarrow{then} T15 \xrightarrow{then} S08 \xrightarrow{then} S10 \xrightarrow{then} S11$.

As mentioned previously, when the number of collisions is greater than N, the server will initiate a negotiation. Client has the privilege to send the required setup configuration to server first. The transition sequence is $S02 \xrightarrow{then} T02 \xrightarrow{then} T17 \xrightarrow{then} S18 \xrightarrow{then} T21 \xrightarrow{then} S20 \xrightarrow{then} S21 \xrightarrow{then} T22 \xrightarrow{then} S19 \xrightarrow{then} T18$. Server can initiate the setup configuration after waiting T_2 time units (delay (T_2) in transition $S22$), if client does not initiate the setup configuration. The transition sequence is $S02 \xrightarrow{then} T02 \xrightarrow{then} T17 \xrightarrow{then} S18 \xrightarrow{then} S22 \xrightarrow{then} T19 \xrightarrow{then} T20 \xrightarrow{then} S23 \xrightarrow{then} S19 \xrightarrow{then} T18$. If server side's requesting negotiation message collides with user's sending data, the client side needs to go to the initial state for preparing the negotiation. The transition sequence is $(S02, T05) \xrightarrow{then} T16 \xrightarrow{then} S17 \xrightarrow{then} T17 \xrightarrow{then} S18 \xrightarrow{then} \ldots$.

In the client side, *Pid* and *Negotflag* are dead variables, and the others are live. In the server side, *Cid* is a dead variable and the others are live. Using the dead and live variables analysis, the number of generated global states is 6891 and there is one type of unspecified reception error in the protocol, which is shown in Figure 7. In contrast, the number of generated global states is 23 381 to detect the unspecified reception error using the exhaustive method. The unspecified reception occurs when server sends a setup message after T_2 time units and client also sends a setup message at the same time. To eliminate the logical error, a new transition is added in each entity. In the client side, transition $T22$:

$$\frac{?SetupReq}{PhaseFlag := 1 - PhaseFlag}$$

is added from state 2 to state 0; in the server side, transition $S24$:

$$\frac{?SetupReq}{PhaseFlag := 1 - PhaseFlag}$$

is added from state 2 to state 0. Additionally, we want the negotiation to restart after collision, so transition $T23$:

$$\frac{NegotFlag = 1 and PhaseFlag = 0}{PhaseFlag := 1 - PhaseFlag; !NegotReq}$$

is added from state 0 to state 1 in the client side, transition $S25$:

$$\frac{?NegotReq, PhaseFlag = 0}{PhaseFlag := 1 - PhaseFlag}$$

is added from state 0 to state 1 in the server side. The adding procedure is shown in Figure 8.

After modification, variable *NegotFlag* is changed from dead to live on the client side. All variables' attributes have no change on the server side. The associated protocol design system will have the dead and live variable analysis, modify the associated global states, and then continue the verification. As a result, some global states need to be re-explored. Using the dead and live variables analysis, the number of global states of the modified protocol is 6959, in which 68 global states are newly generated. Without using incremental processing, all 6959 global states should be re-explored again. Using the exhaustive analysis, the number of global states of the modified protocol is 23 925, in which 644 global states are newly generated.

5. DISCUSSION

The complexity of incremental verification, e.g. $O(k_I * x^{n_I})$, is similar to that, e.g. $O(k_G * x^{n_G})$, of the traditional global state reachability analysis. From theoretic viewpoint, the rate of n_I/n_G is the key point. However, from system users' viewpoint, the rate of K_I/K_G is also important. Unfortunately, the reduction rate of k is always neglected. Considering a system that needs 1000 h to execute an application, no one will use it willingly. However, if the system is improved for 1000 speedup, i.e., 1 h for executing the same application, users definitely may be willing to try the improved system. It is with the K_I/K_G rate that incremental processing is improved. In fact, most of currently existing global state reduction techniques also improve the K_I/K_G rate.

Incremental processing is in fact similar to the pattern match algorithms used in AI's production systems, e.g. the OPS5 production system (Forgy, 1982; Gupta *et al.*, 1988; Acharya and Gupta, 1992; Ishida *et al.*, 1992). In these production systems, incremental tests and incremental processing of pattern match are used. Otherwise, these systems are not able to execute a lot of practical applications. The exploration of global states based on the global state reachability graph is really similar to the pattern match based on the working memory in

production systems. Incremental design especially shows its effectiveness when the modification is small. An open question is whether it is possible to calculate the rate of k_I/k_G. The main difficulty is that it is impossible to calculate the execution time of each statement in a program, which is machine-dependent; and it is very hard to decide how many statements have been executed, which is program-dependent (specification-dependent). Therefore, the invoking of incremental processing is decided by users, i.e. incremental processing provides an optional choice for designers' convenience. In other words, incremental processing can be viewed as a heuristic for protocol design and the designers decide to invoke it optionally.

Two conditions that are suitable to invoke incremental verification process are exemplified as follows: (1) at the final state of protocol design, in which case very few logical errors exist and (2) communication protocols are designed by protocol design experts whose capabilities will result in very few logical errors during protocol design.

In addition to applying the incremental verification method to protocol analysis, we expect that the application domain of this method can be expanded intuitively. That is, some protocol design methodologies can be associated with our incremental verification method. For example, protocol synthesis (Zafiropulo *et al.*, 1980; Rajagopal and Miller, 1991, Shiratori *et al.*, 1991) and stepwise protocol design (Chow *et al.*, 1985; Lin, 1991; Lin and Tarng, 1993) may be possible application domains. At each stage of protocol synthesis or stepwise protocol design, incremental verification can be invoked to analyze the newly generated state space from the previous stage to the current stage.

6. CONCLUSION

Protocol verification is an important issue for computer networking. Global state reachability analysis is one of the most straightforward and easily automated methods for protocol verification. Although global state reachability analysis suffers from the state explosion problem, protocol designers still need to use it. The main reason is that man power verification contributes much more to the cost than computer power does. Therefore, before a better automatic verification technique is proposed, protocol designers still need to use global state reachability analysis to detect logical errors automatically at the early stage of protocol design. Global state reduction techniques and our proposed incremental verification method provide some heuristics to improve the efficiency of global state reachability analysis.

In this paper, we have presented an ECFSM-based protocol verification method. Our method also incorporates the dead and live variable concept to improve efficiency. Both the effects of dead variables becoming live ones and live variables becoming dead ones should be considered at first; then the effect of adding/deleting transitions in the global state reachability graph are identified. Using the proposed ECFSM-based incremental protocol verification method, we have developed an Estelle-based incremental protocol verification system. Thus, an FDT-based incremental protocol design environment is achievable.

ACKNOWLEDGEMENTS

The research is supported by the National Science Council of the Republic of China under the grant NSC 81-0408-E-006-568.

REFERENCES

Acharya, A., Tambe, M. and Gupta, A. (1992) Implementation of production systems on message-passing. *IEEE Trans. Parallel and Distributed Syst.*, **3**, 477–487.

Algayers, B., Coelho, V., Doldi, L., Garavel, H., Lejeune Y. and Rodriguez, C. (1993) VESAR: a pragmatic approach to formal specification and verification. *Comp. Networks ISDN Syst.*, **25**, 779–790.

Anderson, D. P. and Landweber, L. H. (1984) A grammar-based methodology for protocol specification and implementation. In *Proc. ACM 9th Data Communication Symp.*, pp. 63–70.

Belina, F. and Hogrefe, D. (1989) The CCITT-specification and description language SDL. *Comp. Networks ISDN Syst.*, **16**, 311–341.

Brand, D. and Zafiropulo, P. (1983) On communicating finite-state machines. *J. ACM*, **30**, 323–342.

Bochmann, G. v. (1989) Protocol specification for OSI. *Comp. Networks ISDN Syst.*, **18**, 167–184.

Budkowski, S. and Dembinski, P. (1987) An introduction to Estelle: a specification language for distributed systems. *Comp. Networks ISDN Syst.*, **14**, 25–59.

CCTT Recommendation Z.100. (1988) *Specification and Description Language SDL*, AP IX-35.

Chanson, S. T., Loureiro, A. A. F. and Vuong, S. T. (1993) On tools supporting the use of formal description techniques in protocol development. *Comp. Networks ISDN Syst.*, **25**, 723–739.

Chow, C. H., Gouda, M. G. and Lam, S. S. (1985) A discipline for multi-phase communicating protocols. *ACM Trans. Comp. Syst.*, **3**, 315–343.

Chu, P. M. and Liu, M. T. (1989) Global state graph reduction techniques for protocol validation in the EFSM model. In *Proc. IEEE Pheonix Conf. on Computers and Communications*, pp. 371–377.

Courtiat, J. P. (1987) How could Estelle become a better FDT? In *Proc. Protocol Specification, Testing and Verification*, pp. 43–60.

Courtiat, J. P. (1988) Estelle*: a powerful dialect of Estelle for OSI protocol description. In *Proc. Protocol Specification, Testing and Verification*, pp. 171–186.

Courtiat, J. P. and Saqui-Sannes, P. de. (1992) ESTIM: an integrated environment for the simulation and verification of OSI protocols specified in Estelle*. *Comp. Networks ISDN Syst.*, **25**, 83–98.

Diaz, M. (1982) Modeling and analysis of communication and cooperation protocols using petri net based models. *Comp. Networks*, **8**, 419–441.

Forgy, C. L. (1982) RETE: a fast algorithm for many pattern/many object pattern match problem. *Artif. Intell.*, **19**, 17–37.

Frieder, O. and Herman, G. E. (1989) Protocol verification using database technology. *IEEE J. Selected Areas Commun.*, **7**, 324–334.

Frieder, O. (1992) A parallel database-driven protocol verification system prototype. *Software Practice Exp.*, **22**, 245–264.

Gouda, M. G. (1993) Protocol verification made simple: a tutorial. *Comp. Networks ISDN Syst.*, **25**, 969–980.

Gouda, M. G. and Han, J. Y. (1985) Protocol validation by fair progress state exploration. *Comp. Networks ISDN Syst.*, **9**, 353–361.

Gouda, M. G. and Yu, Y. T. (1984) Protocol validation by maximal progress state exploration. *IEEE Trans. Commun.*, **32**, 94–97.

Gupta, A., Tambe, M., Kalp, P., Forgy, C. and Newell, A. (1988) Parallel implementation of OPS5 on the encore multiprocessor: results and analysis. *J. Parallel Program.*, **17**, 95–124.

Horlzmann, G. J. (1993) Design and validation of protocols: a tutorial. *Comp. Networks ISDN Syst.*, **25**, 981–1017.

Huang, C. M., Chang, Y. I. and Liu, M. T. (1990) A computer-aided protocol design by production systems approach. *IEEE J. Selected Areas Commun.*, **8**, 1748–1762.

Huang, C. M., Hsu, J. M., Lai, H. Y., Pong, J. C. and Huang, D. T. (1993) An Estelle interpreter for incremental protocol verification. In *Proc. IEEE 1st Int. Conf. on Network Protocols (ICNP)*, pp. 326–333.

Ishida, T., Gasser, L. and Yokoo, M. (1992) Organization self-design of distributed production systems. *IEEE Trans. Knowledge Data Eng.*, **4**, 123–134.

ISO—Information Processing Systems—Open Systems Interconnection (1987) *ESTELLE—A Formal Description Technique Based on Extended State Transition Model.* DIS. 9074.

Itoh, M. and Ichikawa, H. (1983) Protocol verification algorithm using reduction reachability analysis. *Trans. IECE Jap.*, **E66**, 88–93

Lin, F. J., Chu, P. M. and Liu, M. T. (1987) Protocol verification using reachability analysis: the state explosion problem and relief strategies. In *Proc. ACM SIGCOMM Workshop*, pp. 126–135.

Lin, H. A. (1991) Constructing protocol with alternative functions. *IEEE Trans. Comp.*,, **42**, 376–386.

Lin, H. A. and Tarng C. L. (1993) An improved method for constructing multiphase communication protocols. *IEEE Trans. Comp.*, **42**, 15–26.

Linn, R. J. (1988) *The Features and Facilities of ESTELLE.* National Institute of Standards and Technology.

Linn, R. J. (1989) Conformance testing for OSI. *Comp. Networks ISDN Syst.*, **18**, 203–219.

Liu, M. T. (1989) Protocol engineering. *Adv. Comp.*, **29**, 79–195.

Pehrson, B. (1989) Protocol verification for OSI. *Comp. Networks ISDN Syst.*, **18**, 185–201.

Rajagopol, M. and Miller, R. E. (1991) Synthesis of comunication protocols: survey and assessment. *IEEE Trans. Comp.*, **40**, 468–476.

Shiratori, N., Zhang, Y. X., Tatahsshi, K. and Noguchi, S. (1991) A user friendly software environment for protocol synthesis. *IEEE Trans. Comp.*, **40**, 477–486.

Sidhu, D. P. (1989a) Experience with formal methods in protocol development strategy. In *Proc. Formal Description Techniques II*, pp. 437–453. North-Holland, Amsterdam.

Sidhu, D. P. (1989b) Semi-automatic implementation of OSI protocols. *Comp. Networks ISDN Syst.*, **18**, 221–238.

Umbaugh, L. D., Liu, M. T. and Graff, C. J. (1985) Specification and validation of the transmission control protocol using transmission grammar. In *Proc. IEEE COMPSAC*, pp. 207–216.

West, C. H. (1992) Protocol validation—principles and applications. *Comp. Networks ISDN Syst.*, **24**, 219–242.

Zafiropulo, P. *et al.* (1980) Towards analyzing and synthesizing protocols. *IEEE Trans. Commun.*, **28**, 651–660.