ELSEVIER

Information and Software Technology 42 (2000) 229–244

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Path-based protocol verification approach

W.-C. Liu*, C.-G. Chung

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Da-Sheuh Rd, Hsin-Chu, Taiwan 30050, ROC

Received 17 December 1998; received in revised form 11 June 1999; accepted 20 July 1999

Abstract

Protocol verification is one of the most challenging tasks in the design of protocols. Among the various proposed approaches, the one based on reachability analysis (or known as state enumeration approach) is of the most simple, automatic and effective. However, the state explosion problem is a principle obstacle toward successful and complete verifications of complex protocols. To overcome this problem, we proposed a new approach, the "path-based approach." The idea is to divide a protocol into a collection of individual execution record, denoted as concurrent paths, a partial order representation recording the execution paths of individual entities. Then, the verification of the protocol is, thus, replaced by that of individual concurrent paths. Since concurrent paths can be automatically generated through Cartesian product of the execution paths of all modules, and verified independently, the memory requirement is limited to the complexity of individual concurrent path rather than the whole protocol. Thus, the state explosion problem is alleviated from such "divide and conquer" approach. Furthermore, we propose an algorithm, making the trade-off on memory requirement to generate the concurrent paths more efficiently; and utilize the technique of symmetric verification, parallel computing to improve the efficiency of verification. Eventually, our experience of verifying real protocols shows that our approach uses much less memory and time than reachability analysis.

Keywords: Protocol verification; Concurrent path; Path-based verification

1. Introduction

In the design of protocols, one of the most challenging tasks, the protocol verification is to certify that the protocol executes correctly without crucial logical errors such as deadlock, livelock, channel overflow. Among the various proposed approaches, the one based on reachability analysis (or known as *state enumeration*), a technique to enumerate all the reachable states of a system from an initial one, is of the most simple, automatic and effective [12,25]. However, the technique suffers from the "state explosion problem" [12,25,32]. This problem shows the phenomenon that the number of states grows exponentially with the complexity of the protocol. Quantitatively, a protocol has at most $|Q|^N((|M|+1)^h)^{NN}$ states, where N is the number of units in the protocol, Q the number of local states for each unit, M the number of message types and h the number of channel capacity [19]. Although, the number of reachable states is several orders of magnitude smaller than this amount, that is the number of syntactically reachable states, it still grows exponentially with the complexity of protocols. All reachable states must explicitly or implicitly be memorized in a reachability graph (RG), a directed graph containing all the reachable states in its nodes, to avoid generating duplicate states and to exclude the infinite exploration when cycles exist in the graph. Due to the limited memory capacity, on the basis of Ref. [12], the full search reachability analysis can verify the protocol with 10^5 states, and the most controlled partial search or known as *relief strategies* [21] can be up to 10^8 states, which is intended to reduce the number of global states necessary to be explored. Although, the technique of BDD [4] has made much progress in the number of states [5], the efficient use of BDD depends on the problem domain. The conventional reachability analysis technique outperforms the BDD-based technique in some cases [16] and is still the most general approach for protocol verification. In addition to these methods, other categories of verification approaches such as the *hashing technique* [12,34], *partial ordering* [8,28], *compositional methods* [9,29], and *on-the-fly technique* [7,10,30] provide different contributions for solving the state explosion problem, but they do not contribute complete solutions [21].

Inspecting the reachability analysis technique and its variations, the major hurdle to a successful verification is the enormous size of the RG and the necessity to memorize the complete graph. (Another related problem is the long verification time and it becomes minor if the complete

* Corresponding author. Tel.: + 886-35712121; fax: + 86-35727273.
E-mail address: wcliu@csie.nctu.edu.tw (W.-C. Liu).

0950-5849/00/$ - see front matter
PII: S0950-5849(99)00061-0

verification is necessary and the parallel verification is possible.) However, assuming the RG is available and designating a fixed order among the arcs originating from the same node, we can virtually travel the RG by a systematic depth-first strategy and enumerate all the paths in the RG, i.e. the so-called *execution sequences*, in the forms of simple or cyclic paths as done in the path-based testing of structural testing [1]. Furthermore, if all these execution sequences are generated independently upon the others without constructing the RG, the desired properties of the protocols whose definitions rely upon the global states and their relations can be analyzed from individual execution sequences one by one, i.e. the safety properties and some liveness properties. Accordingly, the memory requirement to remember the global states is limited by the length of an execution sequence rather than the complete RG, and the state explosion problem may be alleviated from such "divide and conquer" approach [21].

To achieve the aforementioned goal to generate execution sequences independently, we found that the concept and generation method of concurrent path used in concurrent program testing [15,27] is a clue. The concurrent path is a partial-order representation, which is a combination of execution behavior of individual execution units, i.e. the unit path of individual execution unit. The set of all concurrent paths is included by the Cartesian product of all unit paths of individual execution unit that are easily generated. The analysis to determine whether a member in the product is a concurrent path is performed independent of other members. Hence, the memory requirement to generate the concurrent path depends on the complexity of individual execution unit and individual concurrent path rather than the execution space of the whole concurrent program. The state explosion problem is thus solved by the introduction of concurrent path in the concurrent program testing [15]. As for the area of protocol verification, the concurrent path can be used to denote the execution behavior of protocols that is originally represented by the execution sequence. In fact, its concept and generation is similar to the pioneering duologue approach by Zafiropulo and West [31,33,36]. However, the duologue approach has the following problems in comparison to that of concurrent path:

1. Limit of entities: the verification methods using duologues, the predicate method of Ref. [36] or the phase diagram of Ref. [31], were limited to two entities. This is the major problem of the duologue-based method.
2. Flood of nonoccurrable duologues: the duologue or concurrent path is an arbitrary combination of unilogue or execution path, but well-behaved duologue or valid concurrent path is not and usually occupies a small amount of all. The nonoccurrable duologue must be removed more efficiently than the well-behaved one is analyzed; otherwise, the performance dramatically aggravates. In their approaches, all types of duologues were analyzed in the same manner.
3. Definition of path: the execution unit of a concurrent program is usually structured and has an explicit exit so that the path is easily defined as a sequence of statements from the entry to the exit of the program. In a protocol, the entry is usually clearly defined as the initial state, but the exit or terminal state does not always exist. In the duologue method, the terminal state was enforced so that the path could be defined. Another problem is about the loops in the path. In a program, the loop is defined by the iteration statement, but the loops in the unilogue was not clearly defined so that not all loops were identified in the unilogues in Ref. [36]. Furthermore, the number of unilogues become infinite if the loop exists. In Ref. [36], an arbitrary limit on the loop time was enforced to avoid the infinite number of unilogues.

Therefore, in this paper, we will concentrate on formalizing the concurrent path of protocols to perform the verification without the above problems. The new approach is called the *path-based approach*. In our approach, the path, the concurrent path and the relations with the conventional execution sequences of protocols are formally defined. The generation and verification methods are also proposed so that the basic properties, such as the absence of deadlock, unspecified reception, livelock and tempo-blocking are verified.

This paper is organized as follows. In Section 2, we overview the crucial concepts of the approach, including the underlying protocol model, the *Communication Finite State Machine* (CFSM) model, the concurrent path, and the basic idea of our approach. In Section 3, we formalize our approach, giving the definition, properties and generation of concurrent paths. In Section 4, we show the steps toward the path-based approach: (a) generating the paths for each FSM and all the concurrent path candidate and (b) verifying the candidates for its validity (since some candidates are invalid because the candidates are arbitrary combinations of paths) and, if it is valid against the properties by performing the reachability analysis on the concurrent path. Specially, in Section 5, we present several performance improvement techniques for our approach. An example protocol is verified using our approach in Section 6. Discussion and comparison with other approaches are presented in Section 7, and the future work is shown in Section 8.

2. Concepts of path-based protocol verification

2.1. CFSM model

The underlying protocol model, in this paper, to prescribe a protocol, is the *Communication Finite State Machine* (CFSM) model, which is a collection of modules communicating with each other via messages. A protocol P in the CFSM model, denoted as a *CFSM system* or shortly a

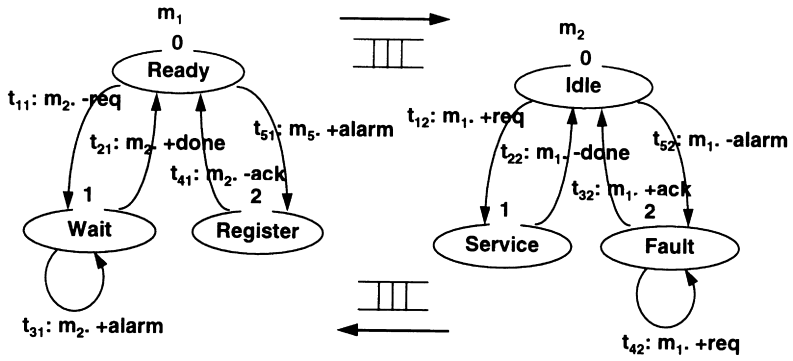

Fig. 1. A simple protocol.

system, is 5-tuple:

$$P = (\langle S_i\rangle_{i=1}^n, \langle M_{ij}\rangle_{ij=1}^n, \langle O_i\rangle_{i=1}^n, \langle Z_i\rangle_{i=1}^n, \langle \Delta_i\rangle_{i=1}^n)$$

where n is the number of modules, i.e. $m_1, \ldots, m_n$; S_i is the set of states of m_i and $S_i \cap S_j = \phi$ for $i \neq j$ ("ϕ" denotes an empty set); O_i and Z_i represent the initial and terminal states of m_i that range over S_i; respectively, M_{ij} is the set of messages that can be sent from m_i to m_j; M_{ii} is empty for each i, and $M_{ij} \cap M_{pq} = \phi$ for $i \neq p$ or $j \neq q$, and Δ_i is a partial mapping function: $S_i \times I_i \rightarrow S_i$, and $\Delta_i(s, x)$ is the state entered after the m_i receives the message x in state s, for each i. ($I_i = \bigcup_{j=1}^n M_{ij}$ is the set of messages that can be received by m_i).Each module m_i in the CFSM system P is a *Finite State Machine* (FSM) composed of states and transitions defined by S_i and Δ_i, respectively. Every two modules m_i and m_j are connected by a dedicated communication channel to transmit the message in M_{ij} from m_i to m_j, which is modeled by an FIFO queue with a channel capacity C_{ij} limiting the number of messages in the channel. (Notice that in the following discussion, the symbols and notions defined in a place retain the same definitions in their succeeding discussions, if they are not especially redefined).

Fig. 1 shows a simple CFSM system with two modules in a graphical form. The label attached to the transition t in m_i in the type of $m_j.+msg$, or $m_j.-msg$ are called the *sending* or *receiving transitions*, respectively, where m_j $(1 \leq j \leq n)$ is the module sending or receiving the message *msg* ($msg \in M_{ji}$ or M_{ij}, respectively). (When there are only two modules, the label of m_i is always the other one and can be omitted.) The transition t is defined by the function $\beta(t) = \Delta_i(\alpha(t), \lambda(t))$, where $\alpha(t)$ and $\beta(t)$ are referred to the heading and tail states of t, respectively, and $\lambda(t)$ denotes the message to be sent or received in transition t.

The status of a CFSM system, at any moment of execution, is depicted by the global state of the system recording the states of the constituent modules and the contents of the communication channels. A *global state* g of the CFSM system P is a pair

$$g = \langle S, C\rangle,$$

where S is a n-tuple of states $\langle s_1, \ldots, s_n\rangle$ (s_i represents the current state of module m_i), and C is a n^2-tuple $\langle c_{11}, \ldots, c_{1n}, c_{21}, \ldots, c_{nn}\rangle$. ($c_{ij}$ is a sequence of messages ranging over M_{ij} whose length is denoted as $|c_{ij}|$. The message sequence c_{ij} represents the contents of the communication channel from module m_i to m_j. Note that every c_{ii} is empty, for M_{ii} is empty.) (The brackets around S and C could be combined without confusion, so that g is in the form of $\langle s_1, \ldots, s_n, c_{11}, \ldots, c_{1n}, c_{21}, \ldots, c_{nn}\rangle$.)

The system stays at a global initial state when it is initialized, and it will finally reach a global terminal state within the normal execution. The *global initial state* of P is a global state, denoted as G_0.[1]

$$G_0 = \langle\langle O_i\rangle_{i=1}^n, \langle\varepsilon\rangle_{i,j=1}^n\rangle$$

(ε denotes an empty message sequence), and the *global terminal state* of P is a global state, denoted as G_T.

$$G_{T'} = \langle\langle Z_i\rangle_{i=1}^n, \langle\varepsilon\rangle_{i,j=1}^n\rangle.$$

The execution behavior of a CFSM system is defined by the relations (called *global transitions* to differentiate with the transitions of modules) between the global states. We define a binary relation "$\Rightarrow$" on global states of P (meaning that P at one global state can be transferred to the other in one step of execution): $g \Rightarrow g'$ iff there exist i, k and x satisfying $s_i' = \Delta_i(s_i, x)$ in either of the following two conditions:

$$c_{ki} = xc_{ki}' \qquad \text{and} \qquad x \in M_{ki}, \tag{1}$$

or

$$c_{ik}' = xc_{ik}, \qquad x \in M_{ik}, \qquad \text{and} \qquad |c_{ik}'| \leq C_{ik}, \tag{2}$$

where $g = \langle S, C\rangle$, $g' = \langle S', C'\rangle$, $S' = \langle s_1', \ldots, s_n'\rangle$, and $C' = \langle c_{11}', \ldots, c_{1n}', c_{21}', \ldots, c_{nn}'\rangle$, and C_{ik}.

The first and the second conditions denote a blocking receiving and a bounded non-blocking sending, respectively. The associative transition with respect to $\Delta_i(s_i, x)$ is referred to the global transition from g to g'.

Extending the relation "$\Rightarrow$", "$\Rightarrow^*$" is the reflexive and transitive closure of "$\Rightarrow$", then a global state g' is reachable from g if $g \Rightarrow^* g'$. g' is said to be reachable with respect to

[1] Capital letters are specially used to represent a specific state, message or information or a set of them.

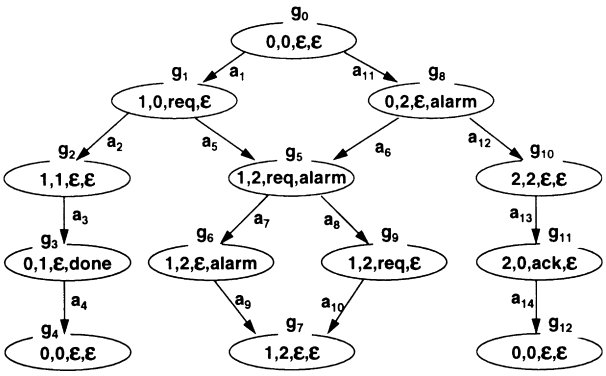

Fig. 2. The reachability graph of the protocol in Fig. 1.

the system if $g = G_0$ and all the global states traversed from g to g' constitutes a *subsequence* of the system.

Note that, examining every reachable global state and subsequence, several types of errors are recognized, such as the *overflow*, *deadlock*, *unspecified reception*, *livelock and temporal blocking* [21]. The channel overflow occurs at a reachable global state when the number of messages in any channels exceeds the capacity of that channel. (In our case, it occurs when condition (2) is to happen, but the limitation of $|c'_{ik}| \leq C_{ik}$ is violated.) A reachable global state $g \neq G_T$ contains a deadlock error if no transition can bring the system to leave this global state and all the channels are empty. The unspecified reception error in any reachable global states is similar to the case of deadlock except that some channels contain at least one message. A global state with any of the above errors is called the *faulty global state*. (When a faulty global state is reached in the execution, we assume no further execution is permitted.) The errors of livelock or tempo-blocking occur when a subsequence has an infinite loop (i.e. a loop without exit) or an undesired loop (i.e. a loop that has exit, but is not a designed one), respectively.[2] Therefore, a *faulty execution sequence* is a subsequence, say $g \Rightarrow^* g'$ containing any faulty global state, livelock, or tempo-blocking. Otherwise, if g is the global terminal state, this subsequence is a *correct execution sequence*. Both the correct and faulty execution sequences are referred to as the *execution sequences* which are the target of verification to explore and analysis.

One simple, but effective method to enumerate all reachable global states and execution sequences, is the reachability analysis technique to demonstrate the interactions among the modules in a total order manner, and to construct the RG as the one shown in Fig. 2. In the RG, every node denotes the global state $g_i = \langle c \rangle$, and every trail from the starting node, such as node g_0 in Fig. 2, to the sink nodes, such as node g_4, g_7 and g_{12}, is an execution sequence, where g_i is the name of the node (global state) and c is the text in the ellipse. For example, there are six trails in Fig. 2, each of which corresponds to an execution sequence as follows:

$es_1 : g_0 \Rightarrow g_1 \Rightarrow g_2 \Rightarrow g_3 \Rightarrow g_4,$

$es_2 : g_0 \Rightarrow g_1 \Rightarrow g_5 \Rightarrow g_6 \Rightarrow g_7,$

$es_3 : g_0 \Rightarrow g_1 \Rightarrow g_5 \Rightarrow g_9 \Rightarrow g_7,$

$es_4 : g_0 \Rightarrow g_8 \Rightarrow g_5 \Rightarrow g_6 \Rightarrow g_7,$

$es_5 : g_0 \Rightarrow g_8 \Rightarrow g_5 \Rightarrow g_9 \Rightarrow g_7,$

and

$es_6 : g_0 \Rightarrow g_8 \Rightarrow g_{10} \Rightarrow g_{11} \Rightarrow g_{12}.$

Among them, es_1 and es_6 are correct execution sequences, whereas es_2, es_3, es_4 and es_5 are faulty ones (they deadlock at g_7.)

However, the reachability analysis technique suffers from the state explosion problem because the size of RG grows exponentially with the complexity of protocol, and thus the conventional reachability analysis technique is not suitable for analyzing complex protocols.

2.2. Basic idea of path-based verification via concurrent paths

Inspecting the reachability analysis and its variations, the major hurdle to a successful verification is the enormous size of the RG and the necessity to memorize the complete graph. However, the properties to be verified depend on the global states and the execution sequence (i.e. the safety properties and some liveness properties). If all execution sequences (thus including all global states) are generated separately without constructing the RG, we can completely verify the system by examining every execution sequence and its global states. The memory requirement to remember the global states is limited by the length of an execution sequence rather than the complete RG, and the state explosion problem may be alleviated from such a "divide and conquer" approach [21].

An execution sequence can be classified as *terminated*, *non-terminated* and *infinite*. A terminated execution sequence is a finite one ranging from the global initial state to the global state in which every module state is at its terminal state, such as es_1 and es_6 in Fig. 2. If we project the sequence of transitions in a terminated execution sequence onto the set of transitions of a module, say m, we can get a sequence of m's transitions and this sequence will be a path of m. (A path of a module is a sequence of transitions whose heading and tailing states are respectively,

[2] We use the positive closure in the regular expression to represent the cycles in an execution sequence. An execution sequence with a cycle, which must be infinite since the cycle can be repeated forever, $g_1 \Rightarrow g_2 \Rightarrow \ldots g_i \Rightarrow \ldots g_j \Rightarrow \ldots (1 \leq i \leq j)$, as $g_1 \Rightarrow g_2 \Rightarrow \ldots (g_i \Rightarrow \ldots g_j)^* \Rightarrow \ldots$, where g_i and g_j are the same global states and denote a cyclic structure. If we represent all the cyclic structure in the infinite execution sequence as above, we can represent the infinite execution sequence in a finite form $g_1 \Rightarrow \ldots (g_i \Rightarrow \ldots g_j)^* \Rightarrow \ldots g_z (1 \leq i \leq j)$, where g_z is the last global state reached, assuming once the terminal global state is reached, no further exploration is allowed.

the initial and the terminal state of that module.) If we perform such a projection of an execution sequence with respect to every module, we can get a set of path each of which owes to a distinct module. For example, the es_1 in Fig. 2, if we project the transitions of es_1, i.e. $[a_1,a_2,a_3,a_4]$ onto the transition of m_1 and m_2, we can get the path $[t_{11},t_{21}]$ of m_1 and $[t_{12},t_{22}]$ of m_2 because $a_1 = t_{11}$, $a_2 = t_{12}$, $a_3 = t_{22}$, and $a_4 = t_{21}$.

As for the non-terminated execution sequence, it has at least one module not reaching its terminal state. If we perform the projection to execution sequence, we can get sequence of subpaths, and the behavior of non-terminated execution sequence can be represented by the set of subpaths. However, for any subpath, it is included by at least one path, provided the terminal state of the module is reachable from every state. (This requirement should be satisfied for any correct system; otherwise, there must be errors of deadlock, livelock or dead code.) Since the execution blocks at the tailing state of the subpath, the additional transitions in the path but not in the subpath are not executable. The behavior of such non-terminated execution sequence can thus also be represented by a set of paths.

An infinite execution sequence must imply that it has cyclic structure as the number of global states is finite; and there exists a last global state (unless the structure of the path is similar to the infinite decimal which should be rare.) Thus, we can also classify the infinite execution sequence into terminated or non-terminated according to the last global state reached and it can also be represented by a set of paths. (These paths may have loops.)

Therefore, we can use a set of module paths, each of which belongs to a distinct module, to represent the behavior of execution sequence. In comparison with the execution sequence, the concurrent path denotes the execution behavior in a partial-order manner, whose ordering relationship is implicitly defined by the precedence of sending and corresponding receiving transitions, and explicitly by the sequential ordering among the transitions in a path [27]; the execution sequence performs in a total-order manner, whose ordering relationship is explicitly defined by the relation of global states. It should be noted that both notations exhibit the same "happen-before" relation of the transitions' execution [20]. Therefore, as long as we can show the behavior of all execution sequences within a RG can be equivalently represented by a set of concurrent paths and we can generate such set of concurrent paths algorithmically, we can use the notation of concurrent path to completely verify the protocol.

The advantage of using the notation of concurrent path instead of the execution sequence is three-fold: (1) all concurrent paths can be generated automatically and under low space complexity; (2) the system is separated into a set of concurrent paths, each of which is much smaller than the original system; and (3) parallel verification. The first advantage results from the representation of the concurrent path, i.e. a set of module paths. The set of all concurrent paths must be included by the Cartesian product of the sets of all module paths. The Cartesian product can be generated easily, but each member of the product is an arbitrary combination of module paths, and not all members are concurrent paths obviously. Thus, as long as we can identify the concurrent paths from Cartesian product we can acquire all concurrent paths. The key of identification results from each member of Cartesian product can be treated as a simplified CFSM system and can be executed. If it is not a concurrent path, it does not denote a real execution behavior and its execution must be blocked at an intermediate global state and render an anomaly of error. (If it can finally reach the terminal global state, it denotes a behavior of terminated execution sequence and is a concurrent path.) Thus, for each member with a blocking global state, if this blocking state is a real one (i.e. no transitions in the original system can release this blocking), it is a concurrent path; otherwise, it is not. We will describe the detail of identification in the next section.

The second advantage is obvious. The original system is now divided into a set of concurrent paths, each of which denotes a partial behavior of and is smaller than the original system. As stated before, each of them can be executed and thus verified independently using the algorithm of the reachability analysis to enumerate the potential execution sequence(s). With these potential execution sequences, we can check the required properties against them. Since each concurrent path is smaller than the original system, the memory requirement of the reachability analysis is also much smaller. Therefore, the state explosion problem is alleviated.

Furthermore, the conventional parallel verification algorithm of Stern and Dill [26] relies on the message passing or shared memory mechanism to build the image of the whole reachability graph. It has from time to time to exchange the newly generated global state among the parallel computers to maintain the consistency of the reachability graph. Thus, a great cost of communication is required. Our approach allows each concurrent path to be verified independently and naturally in parallel. All the information to be exchanged is the verification result of each concurrent path and thus the cost of communication is very low.

In the following sections, we formally define the concurrent path and propose the path-based verification underlying the concept of concurrent path.

3. Definition and generation of concurrent path

3.1. Definition of concurrent path

The concept of concurrent path has been used in the verification and testing of concurrent programs [15,27,35]. The concurrent path defines the execution behavior of concurrent programs by separately specifying the execution path of the execution unit that can execute in parallel, such

as the task in Ada [18]. Similar to the definition of the concurrent path in concurrent programs [15,35], we define the concurrent path prescribing the execution behavior of the CFSM model followed by the definition of paths of a module in the succeeding context.

In order to define the concurrent path of a CFSM system, we first define the *module path*, or briefly *path*.

Definition 1. Within a system P, a finite path p_a in module m_a of length k ($k \geq 0$) is defined as follows: $p_a = [s_1;]$, denoted as ε_a if $k = 0$, or $p_a = [s_1, s_2, \ldots s_{k-1}; t_1, t_2, \ldots, t_k]$ otherwise, where, $s_1 = O_a$and $s_{k+1} = Z_a$, and t_i is the transition from s_i to s_{i+1}, i.e. $\Delta(s_i, \lambda(t_i)) = s_{i+1},\ 1 \leq i \leq k$.

The empty path denoted as ε_a, is also considered as a path and exists only when the initial and terminal states are the same.

Definition 2. Within a system P, an infinite path p_a in module m_a is defined as follows $p_a = [s_1, \ldots, s_i, s_{i+1}, \ldots; t_1, \ldots, t_i, \ldots]$, where $s_1 = O_a$, and t_i is the transition from s_i to s_{i+1}.

In addition to the definitions of paths, we further define the subpaths that are infixes of paths, and prefixed as follows:

Definition 3. An infix or subpath u_a of p_a of module m_a is defined as $u_a = [s_i, s_{i+1}, \ldots, s_{j+1}; t_i, t_{i+1}, \ldots t_j]$ if its length is $j - i + 1 > 0$ or $[s_i;]$ if it is empty, where $1 \leq i \leq j \leq k$.

A nonempty path/subpath $[s_i, s_{i-1}, \ldots, s_{j+1}; t_i, t_{i+!}, \ldots, t_j]$ is written $[t_i, t_{i-!}, \ldots, t_j]$ for short.

Definition 4. A prefix x_a of p_a is a p's subpath whose heading state is the same with p_a and is said to be *included* by p_a, i.e. $p_a = x_a \cdot u_a$, where "$\cdot$" is the concatenation operator upon the subpaths and u_a is a subpath of p_a. If u_a is empty, x_a is denoted as a pure prefix.

Since the number of states and transitions are finite, there must exist cyclic structure in P so that they can appear more than once. Thus, we can introduce the positive closure used in the regular expression [14] to represent the loops the path follows, instead of that in Definition 2.

Definition 5. Within a system P, a nonempty cyclic path p_a in module m_a is defined as $p_a = [s_1, \ldots, (s_i, \ldots s_j)^*, \ldots s_{k+1}; t_1, \ldots, (t_i, \ldots, t_{j-1})^*, \ldots, t_k]$, where $s_1 = O_a, s_{k+1} = Z_a$ and t_i is the transition from s_i to s_{i+1} ($i \geq 1$). The subpath $[s_i, \ldots, s_j; t_i, \ldots t_{j-1}]$ is the loop of p_a.

To differentiate the paths of the above definitions, the paths following Definitions 1 or 2 are called *simple paths*, and that following Definition 5 are *complex paths*. If a simple path does not include any cycle, i.e. for all $i \neq j (1 \leq i, j), s_i \neq s_j$, it is called *pure simple path*.

To define the concurrent path in a CFSM system, we examine the RG and the execution sequences as follows. The finite execution sequences in the RG are divided into two sorts: *terminated* and *non-terminated*. The terminated execution sequence ranges from the global initial state to the global state in which every module state is at its terminal state, such as es_1 and es_6 in Fig. 2. According to the semantics defined in the CFSM model, in a CFSM system with n modules $m_1, m_2, \ldots$, and m_n, with respect to a terminated execution sequence es_{term} of length l ($l \geq 1$) : $g_1 \Rightarrow g_2 \Rightarrow \ldots g_l$, each global transition from one global state to its succeeding one, say $g_i \Rightarrow g_{i+1}$ ($1 \leq i \leq l-1$), requires the execution of only one module transition, say $t_{a_i k_i}$ which is a transition indexed by a_i in certain module m_{ki} ($1 \leq k_i \leq n$). Thus, we can annex the global transition with the module one as $g \overset{t_{a_i k_i}}{\Rightarrow} g_{i+1}$ and rewrite the es_{term} as $g_1 \overset{x_i}{\Rightarrow} g_2 \overset{x_2}{\Rightarrow} \cdots \overset{x_{l-1}}{\Rightarrow} g_l$, where $x_i = t_{a_i k_i} (1 \leq i \leq l-1)$.

For each es_{term}, we can collect the transitions of the module m_k to a transition sequence ts_k each member of which belongs to m_k, i.e. $tS_k = [t_{a_1 k}, t_{a_2 k}, \ldots, t_{a_x k}]$, where $t_{a_i k}$ ($1 \leq i \leq x$) is the transition of m_k, x is the number of m_k's transitions in es_{term}. The ts_k is also a path because of the following reasons:

1. The transition $t_{a_{i-1} k}$ must be the succeeding transition of $t_{a_i k}$ ($1 \leq i \leq x$). This holds because only the transitions starting from the tail state of the last executed transition are possibly executable with respect to the succeeding global state and the module state in the global state remains unchanged until any transition from the same module is executed.
2. The starting state of $t_{a_1 k}$ and the tail state of $t_{a_x k}$ are the initial and terminate states of m_k, respectively, because the es_{term} starts and ends at the global initial and terminal states, respectively.

There is, however, one extreme case that certain module is not involved in the execution of es_{term}, i.e. it always stays at its initial state. Since the null transition sequence ε_p is also considered a path, every module m_i ($1 \leq i \leq n$) must have the corresponding path p_i for es_{term}. (p_i is ts_i if ts_i exists; otherwise, p_i is ε_p). From the module point of view, p_i could be seen as the result of projecting the transition sequence in the es_{term} over the transitions of m_i to denote the corresponding behavior of es_{term} in m_i in terms of paths. Along p_i, each transition is associated with an execution condition to be satisfied, i.e. expecting to send or receive a message. Only when satisfied, the transition is executed and the module is transferred to the tail state.

The complete behavior of the system, in terms of es_{term}, is achieved via the cooperation of all p_i's. On cooperation to render es_{term}, the execution condition of every transition in p_i

will be satisfied, be executed and let the module m_i transfer to its tail state so that the succeeding one will wait for execution. After a series of transitions' executions, m_i finally reaches its terminal state, which is the tail state of the last transition. The system being simplified in a way that every module has only one path, its execution will at least exhibit the execution behavior of the es_{term}. (The behavior of some other execution sequences may emerge because the relative execution speeds of modules are not fixed.) The simplified system is now just a combination of path $\{p_1, p_2,\ldots,p_n\}$ that includes the execution behavior of es_{term} from the system point of view. This combination is denoted as a *concurrent path* for a terminated execution sequence or briefly cp_{term}.

The other sort of finite execution sequence is the *non-terminated execution sequence*, es_{nterm}. This has at least one module not reaching its terminal state, i.e. it must be a faulty execution sequence. As done to the es_{term}, we could collect transitions in the es_{nterm} for each m_k, but there are two situations in the collecting process as follows:

1. If m_k can reach its terminal state in es_{nterm}, the transition sequence collected is a path as in the case of es_{term},
2. If not, the sequence is a pure *prefix* of a path instead of a path.

In situation (2), the execution condition associated with the succeeding transition of the tail state of the last transition in the prefix is not satisfied because of the fault embedded in es_{nterm}. Hence, the system cannot terminate normally in the global terminal state and the transition sequence of m_k is just a pure prefix. If the notation similar to es_{term} is to include the execution behavior of es_{nterm}, the result is a combination of prefixes $\{s_1, s_2,\ldots,s_n\}$, where s_i is a prefix of m_i $(1 \le i \le n)$, and at least one s_i is a pure prefix rather than a path. This combination is still a simplified CFSM system yet with the terminal state of certain module not reachable. If the system is executed, it is also blocked at some global state other than global terminal one.

However, this combination of prefix does not conform to the definition of cp_{term}, but it is reasonable to assume that every state in the original module can reach z_k through a sequence of transitions; that is, z_k could be reachable in the simplified system by adding some transitions between the tail state of $t_{a_x k}$ and z_k. These added transitions have the only function to make the z_k reachable from the syntax point of view, but they will never be executed from a semantic point of view. In other words, the exhibited behaviors of the two systems are the same. Therefore, with respect to the prefix combination $\{s_1, s_2,\ldots,s_n\}$, a path p_i, i.e. the path in the module m_i of new simplified system, can be found to include prefix s_i for every s_i, and the concurrent path $\{p_1, p_2,\ldots,p_n\}$, denoted as cp_{nterm}, instead is used to include the behavior of es_{nterm}.[3] The transitions in p_i but not in s_i $(1 \le i \le n)$, if any, will not be executed and do not influence the behavior of es_{nterm}.

Another sort of execution sequence we have not considered yet is the infinite one. An infinite execution sequence must imply that there exists cyclic structure in it since the number of global states is finite. Thus, an infinite execution sequence $g_1 \Rightarrow g_2 \Rightarrow \ldots g_i \Rightarrow \ldots g_j \ldots (1 \le i \le j)$ can be represented by $g_1 \Rightarrow \ldots (g_i \Rightarrow \ldots g_j)^* \Rightarrow \ldots g_z$ $(1 \le i \le j)$, or $g_1 \Rightarrow \ldots (g_i \Rightarrow \ldots g_z)^*$ $(1 \le i \le j)$, where g_z is the last global state reached assuming once the terminal global state is reached no further exploration is allowed. Thus, the infinite execution sequence can also be classified into terminated and non-terminated, and we can follow the discussions of es_{term} or es_{nterm} to conclude that can be represented by cp_{term} or cp_{nterm}, respectively.

As discussed above, no matter whether es_{term} or es_{nterm} (finite or infinite), its execution behavior is included by the concurrent path (cp_{term} or cp_{nterm}) that is a collection of paths such that there is exactly one path per module in this collection. Expanding this representation of execution behavior, we can define a more general representation form of execution behavior, the *concurrent path*, as follows.

Definition 6. Within a system P, a *concurrent path* is defined as an ordered set of paths $\{p_1, p_2,\ldots,p_n\}$, where p_i is a path of m_i.

The concurrent path denotes the execution behavior in a partial-order manner, whose ordering relationship is implicitly defined by the precedence of sending and corresponding receiving transitions, and is explicitly by the sequential ordering among the transitions in a path [27]; the execution sequence does in a total-order manner, whose ordering relationship is explicitly defined by the relation of global states. It should be noted that both notations exhibit the same "happen-before" relation of the transitions' execution [20]. The notation of concurrent path has the advantage over that of execution sequence that it can be generated in the magnitude of module rather than the RG. Therefore, as long as we can show the behavior of all execution sequences within a RG can be equivalently represented by a set of concurrent paths, and we can generate such a set of concurrent paths algorithmically, we can use the notation of concurrent path to verify completely the protocol with smaller complexity.

In the remaining subsection, we annotate the correspondence relations among both representations to underlie the verification underlying the concept of concurrent path. A concurrent path is said to *correspond* with an execution sequence, and vice versa, if they demonstrate the same execution behavior. The correspondence is notified in Theorem 1.

Definition 7. Given a concurrent path *cp*, the behaviour

[3] It should be noted that the number of such concurrent paths is not limited to one, and any one of them can be used to denote the behavior of $\{s_1, s_2,\ldots,s_n\}$.

set, denoted as $B(cp)$, is a set of all the execution sequences in the reachability graph of *cp*.

Definition 8. A concurrent path *cp* is said to correspond to an execution sequences *es*, if $es \in B(cp)$, denoted as $cp \rightarrow es$.

Lemma 1. *Given a CFSM system P and an execution sequence es, there exists a concurrent path, say cp, so that cp corresponds to es, denoted as* $es \rightarrow cp$.

Proof. The proof has been informally shown in Subsection 2.2, so we omit it here. □

Lemma 2. *For every execution sequence es and concurrent path cp,* $es \in B\ (cp)$ *if* $es \rightarrow cp$.

Proof. Let *es* travels the transitions $t_1, t_2,\ldots,t_n$ in sequence. An $es \in B(cp)$ if

1. all the transitions $t_1, t_2,\ldots$, and t_n are in the union of the transitions of all the paths in *cp*;
2. the sequence $t_1, t_2, \ldots,t_n$ is possible to be executed; and
3. the projection of *es* into the domain of a module is equal to the corresponding path in *cp* of that module. For example, among $t_1, t_2,\ldots,t_n$ ($n(10)$, only t_2, t_4, t_{10} are of module m_i, the projection from *es* to the domain of m_i is the sequence of t_2, t_4, t_{10}.

As *cp* is defined by Definition 7, the conditions (i)–(iii) must be satisfied. □

Theorem 1. *For any execution sequence es, there exists a concurrent path cp such that es corresponds to cp and vice versa, denoted as* $es \leftrightarrow cp$.

Proof. By Lemma 1, a concurrent path *cp* does exist for any execution sequence *es*, and $es \rightarrow cp$. By Lemma 2, $es \in B(cp)$. By Definition 7, $cp \rightarrow es$. Hence, we can find a concurrent path *cp* for any execution sequence *es* such that $es \rightarrow cp$ and $cp \rightarrow es$. □

Since every execution sequence has a corresponding concurrent path, the complete execution behavior is conventionally denoted by the set of all execution sequences and now by the set of all concurrent paths according to Corollary 1 shown below. We can examine all possible execution behaviors by examining all concurrent paths. Once we can generate all concurrent paths, the protocol can be completely verified by being inspected with all the execution sequences of their behavior sets. In the next subsection, we show the method to generate all concurrent paths.

Definition 9. Given a concurrent path *cp*, the behavior set, denoted as $B(cp)$ is a set of all execution sequences corresponding to *cp*.

Corollary 1. *All execution sequences are included by the union of the behavior sets of all concurrent paths*.

Proof. $\forall$ execution sequence *es*, by Theorem 1, $\exists$ a concurrent path *cp*, $es \rightarrow cp$. By Lemma 2, $es \in B(cp)$. □

3.2. Generation of concurrent paths

The purpose of verification is to examine thoroughly the complete behavior of a system against the desired properties. The complete behavior is originally denoted by RG or the set of all execution sequences, and now by the set of all concurrent paths. Ideally, all concurrent paths should be generated for verification. The number of this set of execution sequences following Definition 5 is finite, and so does the corresponding concurrent paths to include these execution sequences by Theorem 1. It is possible to generate all the concurrent paths to represent the complete execution behavior with respect to the issue of number.

One approach to generate all the necessary concurrent paths is the reachability analysis, but obviously is infeasible in its complexity, i.e. confronts the state explosion problem. A new method to reduce the complexity of generation results from the representation of concurrent path. The concurrent path specifies the path that the module will traverse if the CFSM system is performing the corresponding behavior rather than the execution sequences directly specifies the path in the execution space represented by the RG. This gives an idea to generate the concurrent paths from the modules instead of the RG. The representation of concurrent path shows that all concurrent paths are included by the Cartesian product of all the paths of the constituent modules; we can acquire all the concurrent paths from the Cartesian product. (The method to generate paths is available in the area of software testing [1] and is not discussed here.)

However, there are members in the Cartesian product but not having the corresponding execution sequences. A member is denoted as a *concurrent path candidate*, or *candidate* for short. Take the protocol in Fig. 1 as an example. There are at least two paths for each module. The module m_1 has the following two paths:

$$p_{11} = [t_{11}, (t_{31})^*, t_{21}]$$

$$p_{21} = [t_{51}, t_{41}],$$

and the module m_2 has:

$p_{12} = [t_{12}, t_{22}]$,

$p_{22} = [t_{52}, (t_{42})^*, t_{32}]$.

The number of candidates will be four, but there are only six execution sequences in the RG as shown in Subsection 2.1. Thus, we must decide the correspondence between concurrent path candidates and execution sequences, if available. A candidate is said to be *valid* if there is at least one execution sequence corresponding to it; otherwise, it is *invalid*. Consider the candidate $\{p_{21}, p_{12}\}$ and take it as a simplified system. Its execution is always blocked at the global state of $\langle s_0, s_0, \varepsilon, \varepsilon\rangle$ because the execution condition of t_{12} or t_{51} is not satisfied with respect to this global state. Nevertheless, t_{11} ort_{52} can shift the original system from the blocked global state to the new one $\langle s_1, s_0, \varepsilon, req\rangle$ or $\langle s_0, s_2, alarm, \varepsilon\rangle$; thus, this candidate is invalid since the candidate cannot render a valid execution but an anomaly of unspecified reception. As for the concurrent path $\{p_{11}, p_{22}\}$, it can reach the global state $\langle s_1, s_2, \varepsilon, \varepsilon\rangle$ via four different execution sequences and thus a valid concurrent path.

In general, when the execution condition of a transition cannot be satisfied within the candidate but that of another transition can within the original system, the execution of the candidate will render an anomaly of deadlock, or unspecified reception, which will never happen in the real execution. This candidate is classified as an invalid one. The appearance of such invalid candidates comes from those candidates that are arbitrary combinations of paths. The invalid candidate is the combination corresponding to non-real behavior. The invalid candidates raise two issues to be solved:

1. If we directly apply the verification to all candidates, we cannot determine whether it has an anomaly or a real error.
2. The number of candidates is tremendous because it is the product of the numbers of modules' paths, and the verification of each candidate is quite time-consuming via the technique similar to the reachability analysis. If the invalid candidate can be excluded efficiently, the performance of verification will be improved.

Hence, it is necessary to develop a procedure to efficiently remove these invalid candidates. In this subsection, we discuss the removing method and leave the performance issue in Section 5.

Since, a candidate is a CFSM system yet simplified, and according to the semantics of the CFSM model, it can be executed via the reachability analysis to examine whether it renders a real execution. (Reachability analysis technique is adopted because of its simplicity and its capacity to accommodate most protocol underlying models although it is not efficient enough, but the efficiency will be improved later.) A non-real execution must be blocked at some global state that is not a global terminal one and not a blocked one in the system. The blocked state denotes an anomaly of error, and there exists at least one transition in the CFSM system but not the succeeding transition in the candidate to release this blocking situation. By checking the validity of blocking state, we can determine whether a candidate is a concurrent path or not.

The removal of invalid candidates underlies the reachability analysis, and the verification can be embodied into the removal process directly. Therefore, we only need a complement checking procedure into the removal process to complete the entire verification process.

4. Verification in terms of concurrent paths

Based on the idea of concurrent path, we propose a new approach, called the *path-based approach* for the protocol verification based on the CFSM model. This approach involves three steps:

1. enumerate the paths for individual modules;
2. compose the concurrent path candidates; and
3. remove invalid candidates through the reachability analysis and perform the verification.

The first step is to generate the paths of module. The general algorithms to generate the paths can, for example, be found in Refs. [1,20]. In our case, the generation could logically be divided into two phases. At first, two types of the subpaths are generated, the pure simple path and the loops around the loop states. Then, according to the above recommendation, the generated paths are the pure simple ones and complex paths. In Ref. [6], Chang detailed the generation of paths along with the algorithm.

When all the paths are available, the concurrent path candidates are generated through the Cartesian product of all the paths of the constituent modules because the concurrent path candidate is a combination of the paths of the constituent modules. In this subsection, we present a simulation method to determine the validity of candidate.

The simulation follows the semantics of the CFSM defined in Subsection 2.1. Candidates are analyzed with the full search reachability analysis. Since the number of transitions in a path is finite, the simulation of a concurrent path candidate always terminates and has the following results:

1. It is *terminated*, if all transitions on the paths are executed and the last reached global state is the global terminal state.
2. It is *improperly terminated*, if all transitions on the paths are executed, but the last global state is not the global terminal state.
3. It is *blocked*, if there exists at least one transition in a path that cannot be executed because it is a sending transition and the associative channel has reached its capacity, or it is a receiving transition and the heading message in the associative channel is not the necessary one. The global

Candidate		Number of global states visited	Last Global State of Simulation	Result of Simulation	Type of Candidate
1	$\{p_{11}, p_{12}\}$	5	$<s_0, s_0, \varepsilon, \varepsilon>$	terminated	valid
2	$\{p_{21}, p_{12}\}$	5	$<s_0, s_0, \varepsilon, \varepsilon>$	blocked	invalid
3	$\{p_{11}, p_{22}\}$	5	$<s_1, s_2, \varepsilon, \varepsilon>$	blocked	valid (deadlock)
4	$\{p_{21}, p_{22}\}$	5	$<s_0, s_0, \varepsilon, \varepsilon>$	terminated	valid

Fig. 3. Analysis result of the protocol in Fig. 1.

state that the system reaches when the blocking occurs is called the *blocked global state*, and the transitions in the candidate that cannot occur from the blocked global state are collected into a set, called *disabled set*.

4. It is *lively locked*, if there exist loops in the paths and cycle in the corresponding execution sequence.

Both the terminated and improperly terminated candidates are obviously valid. The execution of the terminated one does not contain any actual errors other than the tempo-blocking. The tempo-blocking occurs when there are global states occurring more than once in the corresponding execution sequence. That of the improperly terminated, one must contain the unspecified reception, if we demand that no further execution for each module is allowed when that module reaches its terminal state. The messages yielding the unspecified reception are those remaining in the last reached global state. As for the blocked candidate, it corresponds to either the invalid or the valid with errors depending on the reason of blocking. To distinguish them, we can check the blocked global state with the CFSM system to identify the reason. If the global state is also blocked in the CFSM system, i.e. no transition in the system can be enabled from the global state, the candidate is also valid but with errors. For further checking to identify the type of error, the transitions in the system that are also not executable are inserted into the disabled set of the candidate. The error could be one of the following:

1. an *unspecified reception* if there are remaining message(s) in the blocked global state and there is no sending transition in the disabled set; or
2. a *deadlock* if there is no message in the channels and no sending transition in the disabled set.

If there exist at least one transition in the system to occur from the global state (these transitions are collected into a set, called *enabled set*), there are the following possibilities:

1. It is a *livelock* if every transition in the enabled set has been traveled in current searching execution sequence and is the entry to the cycle in it.
2. It is an *invalid* candidate otherwise because the blocked global state is impossible to occur in the actual execution, i.e. it will leave through any transitions of the enabled set.

It should be noted that the livelock and the tempo-blocking might render a channel overflow when the corresponding cyclic sequence always increases the number of messages in the channel after each cycle.

Take the protocol in Fig. 1 as an example again. The module m_1 of the protocol in has three paths,

$$p_{11} = [t_{11}, (t_{31})^*, t_{21}],$$

$$p_{21} = [t_{41}, t_{51}],$$

and m_2 does:

$$p_{12} = [t_{12}, t_{22}],$$

$$p_{22} = [t_{32}, (t_{42})^*, t_{52}].$$

Thus, there will be four concurrent path candidates. The last reached global states and the results of the analysis are listed in Fig. 3. A deadlock is detected in this simple protocol (the third candidate).

The simulation method to perform validity analysis of a candidate is formally described in Algorithm A1. This algorithm contains two procedures: analyze_cp and check_livelock. Analyze_cp first applies the reachability analysis to find the last global state, and determines the type of candidate following the aforementioned discussion. During the reachability analysis, some necessary global states are kept into the variable q and four sets, i.e. visited state set Q, working state set W, current path P and cycle set C. With the depth-first strategy, q keeps the next global state to be reached. The set of W and Q keeps the states being visited on backtracking and visited, respectively. All visited global states along the current searching path are kept in the set of P to determine whether there is a cycle in the current path. When the execution enters a cycle, a global state along its incoming transition is collected into the set of C. When all concurrent paths are analyzed, an addition procedure check_livelock is used to identify the livelock in the system using this cycle set C.

Algorithm A1: Validity Analysis
Input: $p_1,\ldots,p_N$ /*p_i denote a path of module m_i */
Output: valid, invalid
Description: The algorithm analyzes every concurrent

path candidate $\{p_1,\ldots,p_N\}$ and categorize it into two classes: valid and invalid.

Steps:

Validty_Anlyze();
$C = \{\}$;/* cycle set: a global variable shared to both function*/
begin

for every concurrent path candidate $\{p_1,\ldots,p_N\}$ **do**

Analyze_CP($p_1,\ldots,p_N$)
Check_Livelock();

end;
Analyze_CP($p_1,\ldots,p_N$)
begin
g = global_initial_state;
$P = \{\}$; /* current search path */
$W = \{g\}$; /* working set */
$Q = \{\}$; /* visited state set */
valid = False;

/* reachability analysis upon the candidate */
repeat

pop q from W;
/* on reaching the terminal state, check the tempo-blocking and unspecified reception */
if$q$ is the terminal state **then**

if$C$ is not empty **then**/* at least one cycle is reached */
report the candidate as valid w.r.t. current path P (tempo-blocking);
else
report the candidate as valid w.r.t. current path P;
valid = True;

if all module states in q are terminal states of modules && some channel contents are not empty **then**
report the candidate as valid w.r.t. current path P (unspecified reception);
valid = True;
/* check if q is a visited or to be visited state */
if$q$ is in Q or W **then**
break
update P with respect to q;/* update the path with q */
insert q into Q;/* insert q into visited state set */
/check the execution is blocked */
if no executable transition in $p_1 \cup \ldots \cup p_N$ **then**
/* the execution for current path is blocked */

collect all executable transitions in $m_1 \cup \ldots \cup m_N$ into the set E
if any t in E is not a transition in $(p_1 \cup \ldots \cup p_N)$ **then**
report the candidate as invalid w.r.t. current path P;
else if any transition whose heading state is in q is a sending transition **then**
report the candidate as valid w.r.t. current path P (channel overflow)
valid = True;
else if any channel of q is not empty **then**
report the candidate as valid w.r.t. current path P (unspecified reception)
valid = True;
else

report the candidate as valid w.r.t. current path P (deadlock)
valid = True;
/* explore new global state */
for each executable t in $p_{i1} \cup \ldots \cup p_{iN}$ **do**

if$q$ is in P and (q, t) is not in C **then** /* a new exit to the cycle exists */
insert (q, t) into C;/* save this cycle and its entering transition */
Let g as the global state from q after t; /* $q \stackrel{t}{\Rightarrow} g$ */
push g into W

until W is empty/* no executable states available */
if *valid* = False **then**
classify this candidate as invalid;
else
classify this candidate as valid;
end.
Check_Livelock()
begin
for each distinct q in C do

collect all executable transitions in $m_1 \cup \ldots \cup m_N$ into the set E
if (q, t) is in C for every transition t in E **do**

report the candidate containing q has a livelock;
end.

The above analysis always terminates, because of the finite number of transitions in a path. Assume there are n modules, each of them has at most m paths whose lengths are k in average. The time and memory complexity to check all concurrent paths is $O(m^n \times n^{n\times k})$ and $O(k^n)$, respectively.

This algorithm performs the analysis sequentially, but it is worthy of note that the simulation of a candidate is completely independent of that of one another so that the simulations of different candidates can proceed in parallel, which is discussed later.

5. Performance improved techniques

The performance of the above method largely depends on the number of concurrent path candidates to be checked and the checking time for each candidate. However, a candidate is an arbitrary combination of module paths and checked for its validity independently by simulation. If we can record some useful information about the relation of different candidates to eliminate unnecessary simulation, the performance can be improved. In Section 5.1, we observe there are

candidates being invalid for the same reason, and, once a candidate is identified as invalid, all the other candidates with the same invalid behavior are not necessary to be checked again. Thus, the number of candidates to be checked is reduced. In Section 5.2, we observe that, in the case of a system with several modules having the same FSM behavior, the behavior rendered by the permutation of these modules are de facto equivalent. Thus, all candidates with such equivalent relation have to be checked representatively by any one of them to save the checking effort of the others. In addition, to reduce the number of candidates to be checked, as each candidate is checked independently, we can also use the parallel verification technique to improve the performance as shown in Section 5.3.

5.1. Avoid duplicated checking of invalid candidate

Although, the simulation is an effective method to determine the validity of the candidates as well as to detect errors, a potential drawback is about the efficiency of verification because the candidate is an arbitrary combination of paths, and the invalid may occupy a large percentage of all. In the previous method, each candidate is analyzed independently, but there are invalid candidates that result from the same reason. However, on the detection of an invalid candidate, if we can directly find out other candidates that will also be invalid due to the same reason and mark them invalid immediately, when these candidates are encountered in succeeding analysis, the redundancy to determine their validity is eliminated. Therefore, in this subsection, we present a method accelerating the removal of the invalid candidates by reducing such redundancy.

Consider an invalid candidate $cp = \{p_1, p_2, \ldots, p_n\}$ of a CFSM system with n modules blocked at a non-terminal global state $g = \langle s_1, s_2, \ldots, s_n, c_{11}, \ldots, c_{nn}\rangle$. Let t be one of the succeeding transitions from g of some module m_i whose execution condition is not satisfied due to the invalidity and m_j be the corresponding module with respect to t (i.e. if t is a sending or receiving transition, m_j is the destination or source module of t, respectively). The following observations will help eliminate the redundant simulation by giving special marks to the candidates unnecessary to be simulated, denoted as *redundant candidates*:

1. The invalidity results from the relation of m_i and m_j. Any candidate cp' is also invalid if it also contains p_i and p_j as cp, and the execution can make m_i and m_j reach the state s_i and s_j, respectively. If these candidates are blocked at some other global state earlier than g, they may be valid (but with errors) rather than invalid, but this error will ultimately be detected in the analysis of the candidate that satisfy the execution condition of t. Hence, all such cp's can be marked and ignored directly without affecting the result of verification.
2. The invalidity results from the blocking position in module m_j and m_j. Let t_i and t_j be any transitions whose heading state is s_i and s_j in module m_i and m_j, respectively, not executable from g. Let t be the transition that conclude cp's invalidity because it is executable but not included in cp. Let the subpath q_i and q_j be included by p_j and p_j, and have t_i and t_j as their last transition, respectively. Any candidates $cp' = \{p'_1, p'_2, \ldots, p'_n\}$ whose p'_i and p'_j include q_i and q_j, respectively, it will also be marked because as long as the execution of cp', if possible, reaches s_i and s_j in module m_i and m_j, respectively, the contents of channels between them are the same as c_{ij} and c_{ji}, and t_i and t_j are thus not executable but t is; that is, cp' is invalid. If they can not reach these two states, as in observation (1), the error must be detected in the analysis of other candidates.

Therefore, according to the above two observations, when an invalid candidate cp is detected, any candidate cp' is redundant and marked if cp' includes the paths p'_i and p'_j, p'_i and p'_j are defined as observation (2).

To incorporate these two observations into the verification method, $C(n, 2)$ two-dimensional (2D) matrixes are necessary to keep these marks, where C is the combination function. Each matrix m_{ij} has n_i rows and n_j columns, where n_i and n_j is the number paths in module m_i and m_j, respectively. The marks with respect to cp' is marked in the entry $(\zeta(p'_i), \zeta(p'_j))$ of matrix m_{ij}, where $\zeta(p)$ denotes the index of path p in its module. A new candidate $\{p''_1, p''_2, \ldots, p''_n\}$ will be simulated if the entry $(\zeta(p''_i), \zeta(p''_j))$ of matrix m_{ij} is yet not marked, for each i and j $(1 \le i < j \le n)$.

It is difficult to estimate how efficiently the method helps remove the invalid candidates. To one extreme, it may provide no help when the paths of a module are completely independent with other. However, our experiment on the verification of the X.21 protocol shows that, 57% of candidates are marked in advance before they are analyzed.

To improve the efficiency of verification, we make some trade-off on the memory requirement in this performance improvement technique. The original memory requirement is used to keep the global states on the current search trace and the set of cycle entering state, which depends on the length of path. Currently, we add the additional memory requirement on the matrixes to keep the marks on the redundant candidates. Since each entry in the matrix occupies only 1 bit, the total memory to these table are $C(n, 2) \times p^2$ bits, where n is the number of modules and p is the number of paths in a module on the average. Assume there are 16 mega bytes of memory available. To an extent, one can at most verify a protocol with 100 modules each of which has at most 160 paths, or, to the other extent, with two modules each of which has at most 4000 paths. When the number of paths exceeds this limit, the technique of parallel verification (discussed below) can be used.

However, the size of required memory is computed in advance as long as the paths of modules are enumerated. When the memory requirement exceeds the limit of physically available memory, the paths of each module can be split into k independent sets ($k > 1$), each of which includes

$1/k$ portions of paths in the module so that the memory requirement for each verification is down to $1/k^2$. The value of k is chosen to satisfy the available memory capacity. k^p verifications are necessary for every possible combination of the sets of different modules. However, the memory requirement is k^2 times smaller than the original one.

5.2. Symmetric verification

One reason to increase the verification time is the huge number of concurrent path that is the product of the numbers of paths of all modules. In the case of replicated modules (the entities that are represented by the same finite state machine), the concept of symmetric verification [17] is helpful in improving the efficiency of verification. The symmetric verification aims to exploit the structural symmetry in the protocol. The structural symmetries induce an equivalence relation between states. Thus, for each equivalence set induced by symmetry, only one state has to be explored and the number of states to be explored is reduced greatly.

To apply this concept, we must define the symmetry with respect to concurrent paths (or candidates). The symmetry results from the permutation of the paths from the identical modules. (A module is said to be identical to the other if their corresponding finite state machines are the same.) A concurrent path may be almost the same with the other except for some paths from the identical modules are in different order. In this case, they are symmetric and correspond to the same behavior. For example, for the concurrent path $cp = \{p_1, \ldots, p_i, \ldots, p_j, \ldots, p_n\}$ and $cp' = \{p_1, \ldots, p'_i, \ldots, p'_j, \ldots, p_n\}$, if m_i and m_j are the identical modules, p_i and p'_j are the same paths, and p_j and p'_i are two concurrent paths that are symmetric. Thus, for the set of symmetric concurrent paths, we only have examined any one of them instead of all.

Formally, a concurrent path $cp = \{p_1, p_2, \ldots, p_n\}$ is said to be symmetric to another $cp' = \{p'_1, p'_2, \ldots, p'_n\}$, denoted as $cp \backsim cp'$, if, for each i $(1 \le i \le n)$, p_i *and* p'_i are either the same or there must exist another j $(1 \le j \le n$ and $j \ne i)$ such that

1. m_i and m_j are identical, and
2. p_i and p'_j are the same paths.

To explain how to do this, we first assume only the first m $(1 < m < n)$ modules that are identical, and the order of index to the paths in these identical modules are the same (i.e. a path p_i are the same of another path p_j iff the $\zeta(p_i) = \zeta(p_j)$, i.e. the index of p_i is equivalent to that of p_j). Thus the set of symmetric concurrent paths with respect to a concurrent path $\{p_1, \ldots, p_n\}$ can be denoted as

$$\{\{p'_1, \ldots, p'_m, \ldots, p'_n\} | \zeta(p'_1, \ldots, p'_m) \in \pi(\zeta(p_1, \ldots, p_m))\}$$

where ζ is an extension of the original definition of the index function ζ to map the order set of path to its order set of path index, i.e. $\zeta(p_1, \ldots, p_m) = (\zeta(p_1), \ldots \zeta(p_m))$, and π denotes a permutation function over the order set $\zeta(p_1, \ldots p_m)$, that is $\zeta(p'_1, \ldots, p'_m) \in \pi(\zeta(p_1, \ldots, p_m))$ iff for each i, there exist j $(j \ne i)$ such that $\zeta(p_i) = \zeta(p'_j)$.

Then, for the set of symmetric concurrent path candidates, we can add the following rule to check only the first concurrent path candidate reached among them when checking the validity of concurrent path candidate.

Rule: A candidate $\{p_1, p_2, \ldots, p_n\}$ has to be checked if, for each p_i $(1 < i \le m)$, $\zeta(p_i) > \zeta(p_j)$ for all $j < i$.

In summary, with the symmetric technique, instead of verifying all k^n candidate, we only have to verify $k^{(n-m)*} \sum_{i=0}^{n-m+1} (k - i) \approx k^{(n-m+1)*}(n - m + 1)$ candidates, (assume every module has k paths in average).

5.3. Parallel verification

One of the best features of the proposed approach is the verification of different candidates that can be performed independently, and the algorithm can be naturally adapted to be parallel. For example, if N ($N(1)$ groups of candidates are to be verified in parallel, the paths of the module with the fewer number of simple paths is chosen for separation. Its paths are divided into N subsets so that the paths in the same subset have the most similar prefix. Then, N groups of candidates can be generated by the Cartesian product of one subset and the sets of the paths of other modules. Each verification may also maintain their own matrix used in Subsection 5.1 considering all only the paths in its subset. If some information sharing mechanism such as the share memory exists, all the paths are added into the matrix to avoid all redundancies.

6. An example: the verification of X.21 protocol

The X.21 protocol [13] shown in Fig. 4 is used to demonstrate our approach embodied in a verification tool, called the path-based protocol analysis tool, developed in the environment of Windows© 95 on IBM© PC equipped with the CPU of Intel© Pentium and 32 MB of memory. This tool provides the techniques of Sections 4 (verification in terms of concurrent path), 5.1 (avoiding redundant checking) and 5.2 (symmetric verification). We also implement the complete RA in our tool.

The X.21 protocol has two modules, called DTE and DCE, each of which has 24 states and 61 transitions (the number of transition is more than the visible one in Fig. 4 result from the state "all" denoting any other state in the module). The initial state and the quasi-terminal are state 1. (Since the protocol can repeat forever, there is no apparent terminal state and we assume state 1 is a quasi-terminal state that is after state 1 is reached, a new service is going to be provided. The concept of quasi-terminal state will be explained in more detail later.) For each module, it can be decomposed into 53 independent complex paths, and four loops around three distinct states. Thus, we can generate

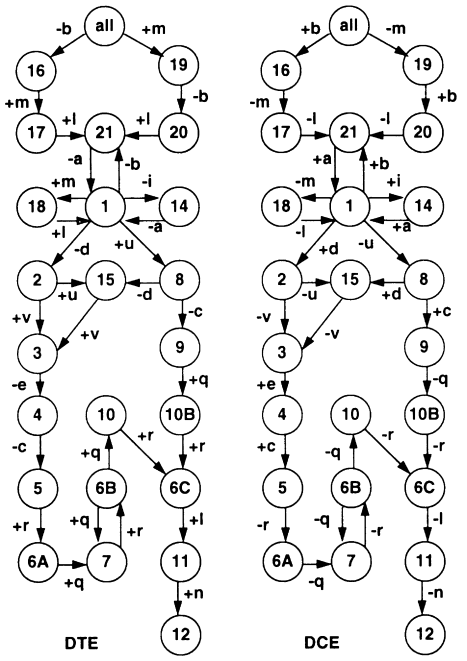

Fig. 4. Specification of X.21 protocol.

2809 concurrent path candidates, and only 221 are valid. There are one deadlock, seven unspecified reception, and 12 tempo-blocking as shown in Fig. 5. The faulty global states with the error of unspecified reception are ⟨16,21,ε,ε⟩, ⟨16,21,b,l⟩, ⟨116,21,b,ε⟩, ⟨16,21,ε,l⟩, ⟨16,3,b,v⟩, ⟨16,3,b,ε⟩, ⟨20,3,b,v⟩. These errors result from the nondeterministic behavior specified in the module, such as for a state there may be two transitions, one is visible in the figure, and the other is to the succeeding state of state

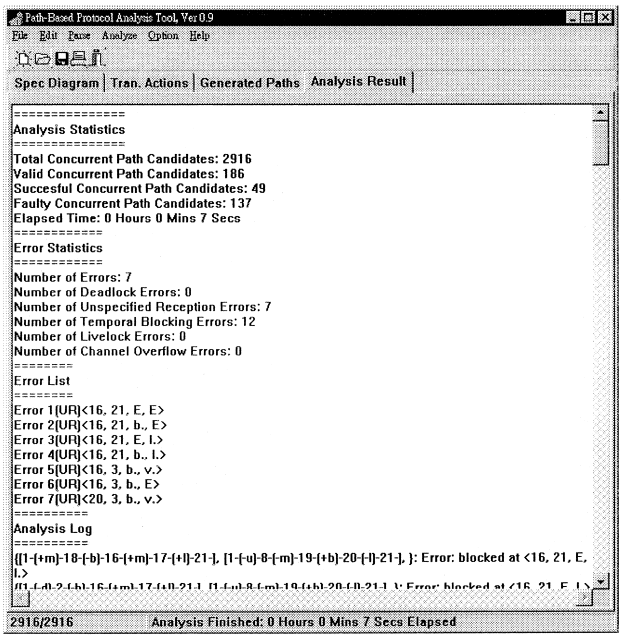

Fig. 5. Result of analyzing X.21.

"all." As for the errors of tempo-blocking, they are due to the complete matched loops between DTE and DCE modules such as the loops between states "1" and "21".

In comparison with the complete RA, the path-based approach take longer time (7 s) than RA (5 s), but the memory requirement of the former is only about 26K at most (the memory for analyzing each concurrent path depends on the length of paths, thus it ranges from 0.9 to 26K and the average is 5K) and that of RA is about 500K. Furthermore, among the 2809 candidates, if we use the technique in Section 5.1, we only have to check 479 candidates actually with the efficiency improvement ratio of 82% and the verification reduce to 2 s.

7. Discussion

7.1. Quasi-terminal state

One potential and crucial drawback is the requirement of terminal state for each module which does not always exists in the protocol, especially in the module that continuously provides the service. Alternatively, there is a quasi-terminal state where the module finishes its last service and is to provide the new. Such state is instead used as a terminal state to generate the paths. The problem on this is that when the system reaches such a state, it may not shortly reach the global terminal state but is blocked at other global state. If this global state has at least one executable transition starting from the quasi-terminal state, it does possibly not denote an error, but the information of the previous service retains to affect the succeeding. Therefore, in this case, when a candidate is blocked at a global state that is not the global terminal one but has at least one module reaching its quasi-terminal state and any transition starting from the quasi-terminal state is executable, further verification is required. All such last global states are kept in a set, and another verification process starts for each new member in this set until no more new member is available or the new one is an unusual state denoting potential errors. As the number of possible last global states is finite and usually small, the verification will finally terminate. With such a technique, only the quasi-terminal state is required that exists in most protocols.

7.2. Comparison with other methods

The study of protocol verification started as early as 1970s [2,24]. Many proposed approaches endeavored to overcome the obstacle of state explosion. In addition to the duologue approach, there are some works that also alleviates the state explosion problem from the memory point of view:

1. *Bounded depth-first search* [12]: this is a depth-first search with a bound on the length of the execution sequences. Its maximum memory requirement is also

the length of an execution sequences. It can do the exhaustive search if the long execution time is allowed. However, it does the double work since no searching history of other execution sequences is kept. It is also difficult to set an accurate upper bound for search without exploring the unnecessary global states.

2. *Protocol expression* [11,23]: this approach describes the behavior of the FSMs with an extended type of regular expression, termed "protocol expression". A cross product and algebraic reduction and equivalence rules, denoted as soundness rules, are used to analyze and reduce the state, which can be performed either by simple manual algebraic analysis or automated cross product evaluator. This approach is a variation of the reachability analysis using the soundness rule. However, the soundness rule can be used only in the CFSM and is difficult to extend to other more complex modules. Besides, the performance, this approach depends on the application of the operators " + " or "*". If these two operators flood, the state space problem emerges.
3. *Tree protocol* [3]: this method grows the execution tree of a module rather than the execution tree of system, thus the memory requirement is limited to the complexity of single modules. However, the rule to grow the tree is quite complicated and also difficult to extend.
4. *On-the-fly* [7,10,30]: the on-the-fly technique use the depth-first search technique and only keep the states along current search path to avoid the state space problem. Since a global state may appear in different search paths, it also use the state cache to keep some forgotten states to avoid wasting verification time in checking the same global states. Thus, the performance of this technique depends on the number of states in the cache. However, our method requires the same amount of memory but uses a simple and small matrix and an efficient checking scheme to avoid redundantly checking the same path.

Therefore, the above approaches do not completely solve state space problem even from the memory point of view. On the basis of Ref. [12], no current state-enumeration approach can completely and efficiently verify the protocol with more than 10^8 global states, but protocols will become more and more complex due to the increasing demand of protocol services. The problem will emerge largely and become more and more difficult to be solved.

8. Conclusion

Our approach is similar to the approach of Zafiropulo [36] but without all the limitations in his work. It underlies the concept of concurrent path dividing the protocol into small and independent components, the concurrent paths. The concurrent path specifies the execution behavior in a partial order manner, whose ordering relationship is implicitly defined. Our approach first generates all the necessary paths of a module, and uses the Cartesian product to generate the concurrent path candidates, then applies the reachability analysis to the candidate to determine the validity of the candidate. Within the reachability analysis of individual candidates, all the logical errors, if any, in the protocol are thus detected. Assume there are n modules, each of which has at most m paths whose lengths are k on average. The time and memory complexity for verifying general concurrent paths are $O(m^n \times n^{n\times k})$ and $O(k^n)$, respectively and can be further reduced to $O(m^n \times n \times k)$ and $O(n \times k)$ for simple concurrent paths, respectively.

To improve the efficiency of the verification, we make some trade-off on the memory requirement to avoid redundant analysis. $C(n, 2)$ 2D matrixes are necessary to mark the redundant candidates that will be invalid due to the same reason of some previously analyzed invalid candidate or a valid but erroneous candidate that will ultimately be detected in the analysis of other candidates, where n is the number of modules. By eliminating the time to analyze these redundant candidates, our experiments shows that the efficiency is improved greatly, but the memory requirement is only $O(n^2 \times p^2)$, where p is the number of paths of a module in average. Therefore, the state explosion problem is completely conquered from the memory point of view with almost the same magnitude of performance of the reachability analysis. Furthermore, if the parallel verification technique is used, our approach can be much faster than reachability analysis.

References

[1] B. Beizer, Software Testing Techniques, 2nd, Van Nostrand Reinhold, New York, 1990.
[2] G.v. Bochmann, C.A. Sunshine, Formal methods for protocol specification and validation, in: C.A. Sunshine (Ed.), Computer Network Architecture and Protocols, 2nd, Plenum Press, New York, 1989 chap. 17.
[3] D. Brand, P. Zafiropulo, On communicating finite-state machines, JACM 30 (1983) 321–324.
[4] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. 35 (1986) 677–691.
[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inf. Comput. 98 (1992) 142–170.
[6] J.M. Chang, An Path-based Protocol Validation Approach, MS thesis, National Chiao Tung University, Taiwan, ROC, 1994.
[7] J.-C. Fernandez, L. Mounier, C. Jard, T. Jéron, On-the-fly verification of finite transition systems, Formal Meth. System Design (1993) 251–273.
[8] P. Godefroid, D. Peled, M. Staskauskas, Using partial-order methods in the formal validation of industrial concurrent programs, IEEE Trans. Software Engng 22 (1996) 496–507.
[9] O. Grumberg, D.E. Long, Model checking and modular verification, ACM Trans. Program. Languages Systems 16 (1994) 843–871.
[10] G.J. Holzmann, A theory for protocol validation, IEEE Trans. Comput. 31 (1982) 730–738.
[11] G.J. Holzmann, Tracing protocols, AT&T Tech. J. 64 (1985) 2413–2433.
[12] G.J. Holzmann, Design and Validation of Computer Protocols, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[13] G.J. Holzmann, Tutorial: design and validation of protocols, AT&T Technical Report TR157, May 1991b.
[14] J.H. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[15] S.Y. Hsu, C.G. Chung, A heuristic approach to path selection problem in concurrent program testing, in: R.-M. Yang (Ed.), Proceedings of the Third Workshop on Future Trends of Distributed Computing SystemsIEEE Computer Society, Los Alamitos, CA, 1992.
[16] A.J. Hu, D.L. Dill, Efficient verification with BDDS using implicitly conjoined invariants, Proceedings of Computer Aided Verification, Fifth International Conference, CAV '93, Lecture Notes in Computer Science, 697, Springer, Verlag, 1993.
[17] C.N. Ip, D.L. Dill, Better verification through symmetry, Formal Meth. System Design 9 (1996) 41–75.
[18] ISO, Information technology—Programming languages—Ada, ISO International Standard ISO/IEC 8652:1995, 1994.
[19] Y. Kakuda, Y. Takada, T. Kikuno, On the complexity of protocol validation problems for protocols with bounded capacity channels, IEICE Trans. Fundam. Electron. Commun. Comput. Sci. E77 (1994) 658–667.
[20] L. Lamport, Time, clocks, and the ordering of events in a distributed system, CACM 21 (1978) 558–565.
[21] M.T. Liu, Protocol design: redefining the state of the art, IEEE Software 17 (1992) 17–22.
[23] A. Lombardo, S. Palazzo, An extended algebra for the validation of communication protocols, Software Engng J. 4 (1989) 148–158.
[24] B. Pehrson, Protocol verification for OSI, Comput. Networks ISDN Sys. 18 (1989/) 185–201.
[25] F. Pong, M. Dubois, Verification techniques for cache coherence protocols, ACM Comput. Surv. 29 (1997) 82–126.
[26] U. Stern, D.L. Dill, Parallelizing the Murphi verifier, Proceedings of the Computer Aided Verification CAV '97, Lecture Notes in Computer Science, 1254, Springer, Berlin, 1997.
[27] K.C. Tai, R.H. Carver, Testing of distributed programs, in: A. Zomaya (Ed.), Parallel and Distributed Computing Handbook, McGraw-Hill, New York, 1996.
[28] A. Valmari, Stubborn Sets for Reduced State Space Generation, in: G. Rozenberg (Ed.), Advances in Petri Nets 1990, Lecture Notes in Computer Science, 483, Springer, New York, 1991.
[29] A. Valmari, Compositional in state space generation, Proceedings of the Advances in Petri Nets 1993, Lecture Notes in Computer Science, 674, Springer, Berlin, 1993.
[30] A. Valmari, On-the-fly verification with stubborn sets, Proceedings of Computer Aided Verification, Fifth International Conference, CAV '93, Lecture Notes in Computer Science, 697, Springer, Berlin, 1993.
[31] C.H. West, An automated technique of communications protocols validation, IEEE Trans. Commun. 26 (1978) 1271–1275.
[32] West, C.H., Protocol validation in complex systems in: Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing, Austin, Texas, 1989.
[33] C.H. West, P. Zafiropulo, Automated validation of a communications protocol: the CCITT X.21 recommendation, IBM J. Res. Dev. 22 (1978) 60–71.
[34] P. Wolper, D. Leroy, Reliable hashing without collision detection, Fifth International Conference on Computer-Aided Verification CAV '93, Lecture Notes in Computer Science, 697, Springer, Berlin, 1993.
[35] R.D. Yang, A Path Analysis Approach to Concurrent Program Testing, PhD thesis, National Chiao Tung Univ., Taiwan. ROC, 1990.
[36] P. Zafiropulo, Protocol validation by duologue-matrix analysis, IEEE Trans. Commun. 26 (8) (1978) 1187–1194.