Symbolic path-based protocol verification

Wen-Chien Liu*, Chyan-Goei Chung

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

Received 13 March 1999; received in revised form 11 June 1999; accepted 20 July 1999

Abstract

The principal problem in protocol verification is state explosion problem. In our work (W.C. Liu, C.G. Chung, Path-based Protocol Verification Approach, Technical Report, Department of Computer Science and Information Engineering, National Chiao-Tung University, Hsin-Chu, Taiwan, ROC, 1998), we have proposed a “divide and conquer” approach to alleviate this problem, the path-based approach. This approach separates the protocol into a set of concurrent paths, each of which can be generated and verified independently of the others. However, reachability analysis is used to identify the concurrent paths from the Cartesian product of unit paths, and it is time-consuming. Therefore, in this paper, we propose a simple and efficient checking algorithm to identify the concurrent paths from the Cartesian product, using only Boolean and simple arithmetic operations. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Protocol verification; Reachability analysis; Path-based approach

1. Introduction

Communication protocols are concurrent software to maintain, coordinate and govern the interactions of concurrent processes in a distributed system. As protocols become increasingly large-scale and complex, the designs of correct protocols are becoming challenging and difficult tasks. To deal the complexity and difficulty of protocol design, protocol verification is introduced. It is a process to verify protocols which are modeled as finite state machines with respect to the crucial logical properties, such as deadlock, livelock, channel overflow, etc. Most verification approaches to date are based on reachability analysis (or known as state enumeration) to enumerate all the reachable states from an initial one and to check whether all reachable states can satisfy the necessary properties [2–4]. Although such technique is simple, automatic and effective, it suffers from the “state explosion problem” [4–6]. This problem asserts that the number of states grows exponentially with the complexity of protocol. All reachable states must explicitly or implicitly be memorized in a reachability graph (RG) to avoid generating duplicate states and to exclude the infinite exploration. Due to the limited memory capacity, on the basis of Ref. [3], the limit of the fully-search reachability analysis is $10^5$ states, and that of the controlled partial search or known as relief strategy [3] is $10^8$ states, which intends to reduce the number of global states necessary to be explored. Although the technique of BDD [8] has mad much progress in reducing the number of states [9], the efficient use of BDD depends on the problem domain, and the conventional state enumeration technique outperforms the BDD-based technique in some cases [10] and is still the most general approach for protocol verification.

The state explosion problem raises two issues: large memory space and long verification time. For the first issue, we have proposed the path-based approach [1] underlying the concept of concurrent path to represent the execution behavior of the protocol into a partial-order representation as a set of unit paths [11,12]. This approach treats the protocol as a set of concurrent paths so that each concurrent path can be generated and verified independently of the others. Thus, the memory required to store reachable global states depends on the complexity of a concurrent path rather than the whole protocol, and the memory space issue is alleviated. However, the issue of long verification time does not have a very good solution. Since, in our approach,
the concurrent path is generated via the Cartesian product of unit paths, each product member (denoted as concurrent path candidate) is an arbitrary combination of unit paths and is not always a concurrent path. We use reachability analysis to identify the concurrent paths from the candidates by checking whether a candidate corresponds to a real execution behavior in the protocol. Although, we use several techniques to avoid the unnecessary check, reachability analysis is still a time-consuming technique to perform such a check.

In Ref. [1], reachability analysis is used to identify the last reachable global state with respect to a concurrent path candidate. As long as the last state is identified, we can classify the candidates as valid or invalid by checking if the last reachable global state is a real blocking state in the system. If it is not, this candidate is identified as invalid. However, for a communication finite state machine (CFSM) model, the execution completely depends on the message types and orders of sending and receiving. Thus, in this paper, we propose an approach to compute the last reachable global state using such relations. The computation uses simple Boolean and arithmetic operations, and the checking speed has improved a lot. The remaining context is organized as follows: In Section 2, we first overview the underlying protocol model, the CFSM model and the path-based approach [1]. Then in Section 3, we illustrate the new method to compute the last reachable state. Finally, we give a conclusion about our work.

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.

**Definition 1.** A protocol $P$ in the CFSM model, denoted as a CFSM system or shortly a system, is 5-tuple:

$$P = \left\{ S_1^n, M_i^n, O_i^n, Z_i^n, \Delta_i^n \right\},$$

where $n$ is the number of modules, i.e. $m_1, \ldots, m_n$, $S_i$ the set of states of $m_i$ and $S_j \cap S_j = \phi$ for $i \neq j$ (“$\phi$” denotes an empty set), $M_{ij}$ the set of messages that can be sent from $m_i$ to $m_j$, $M_{ij}$ the empty for each $i$, and $M_i \cap M_{pq} = \phi$ for $i \neq p$ or $j \neq q$, and $O_i$ and $Z_i$ represent the initial and terminal states of $m_i$ that range over $S_i$, respectively, $\Delta_i$ is a partial mapping function: $S_i \rightarrow S_i$, and $\Delta_i(x, y)$ is the state that entered after the $m_i$ receives the message $x$ in state $s$, for each $i$. ($I_i = \bigcup_{j=1}^n M_{ji}$ 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 (FMSM) composed of states and transitions defined by $S_i$ and $\Delta_i$, 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 a FIFO queue with channel capacity $h_{ij}$ limiting the number of messages in the channel.

Fig. 1 shows a simple CFSM system with two modules in a graphical form. The label attached to the transition $t$ in $m_i$ is of the form $m_i - \text{msg}$, or $m_i + \text{msg}$ for the sending and receiving transitions, respectively, where $m_i$ (1 \leq j \leq n) is the module sending or receiving the message msg ($msg \in M_{ij}$ or $M_{ji}$, respectively). (When there are only two modules, the label of $m_i$ always denotes the other module and can be omitted.) The transition $t$ is defined by the function $\beta(t) = \Delta_i(a(t), \lambda(t))$, where $a(t)$ and $\beta(t)$ are referred to as 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 status of the constituent modules and the contents of the communication channels.

**Definition 2.** A global state $g$ of the CFSM system $P$ is a pair $g = \left( S, C \right)$, where $S$ is a $n$-tuple of states $(s_1, \ldots, s_n)$ ($s_j$ represents the current state of module $m_j$), and $C$ is a $n^2$-tuple $(c_{11}, \ldots, c_{1n}, c_{21}, \ldots, c_{nn})$. ($c_{ij}$ 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 $c_{ij}$ is empty since $M_{ij}$ is empty.) (The brackets around $S$ and $C$ could be combined without confusion so that $g$ is in the form of $(s_1, \ldots, s_n, c_{11}, \ldots, c_{1n}, c_{21}, \ldots, c_{mn})$.)

The system stays at a global initial state when it is initialized and finally reaches a global terminal state on normal execution.

**Definition 3.** The global initial state of $P$ is a global state, denoted as $G_0$, $G_0 = \left\{ (O_1^n, (e)_1^n) \right\}$ ($e$ denotes an empty message sequence). The global terminal state of $P$ is a global state, denoted as $G_T$, $G_T = \left\{ (Z_1^n, (e)_1^n) \right\}$.

The execution behavior of a CFSM system is defined by the relation (called global transitions to differentiate with module transitions) between the global states.
Definition 4. A global transition is defined as 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 \( s'_i = \Delta_x(s_i, x) \) in either of the following two conditions:

\[
e_{ki} = xc_{ki} \quad \text{and} \quad x \in M_{ki}
\]

or

\[
e'_{ik} = \begin{cases} e_{ik} & \text{if } x \in M_{ik} \text{ and } |e_{ik}| < h_{ik} \\ e_{ik} & \text{if } x \in M_{ik} \text{ and } |e_{ik}| = h_{ik} \end{cases}.
\]

where \( g = \langle S, C \rangle, \quad g' = \langle S', C' \rangle, \quad S' = \langle s'_i \rangle_{i=1}^n \quad \text{and} \quad C' = \langle c'_{ij} \rangle_{j=1}^n \).

The first and the second conditions denote a blocking receiving and a non-blocking sending (when the channel reaches the channel capacity, the next sending action can be executed, but its messages are ignored and the channel content remains the same), respectively. The transition associative with \( \Delta_x(s_i, x) \) is referred to as the global transition from \( g \) to \( g' \).

Definition 5. A global state \( g' \) is reachable from \( g \) if \( g \Rightarrow^* g' \), where \( \Rightarrow^* \) is the reflexive and transitive closure of \( \Rightarrow \). \( g' \) is said to be reachable with respect to the system if \( g = G_0 \).

Definition 6. If \( g' \) is reachable from \( g \), then all the global states traversed from \( g \) to \( g' \) constitute a subsequence of the system. If \( g = G_0 \), then the subsequence from \( g \) to \( g' \) is an execution sequence.

One simple but effective method to enumerate all reachable global states and execution sequences, is reachability analysis which demonstrates the interactions among the modules is a total order manner and which constructs the RG as the one shown in Fig. 2. In the RG, every node denotes the global state \( g_i = \langle c_i \rangle \), and every trail from the starting node, such as node \( g_0 \) in Fig. 2, to a sink node, such as nodes \( g_4, g_7 \) and \( g_{12} \), is an execution sequence, where \( c_i \) 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:

- \( x_1: g_0 \Rightarrow g_1 \Rightarrow g_2 \Rightarrow g_3 \Rightarrow g_4 \)
- \( x_2: g_0 \Rightarrow g_1 \Rightarrow g_5 \Rightarrow g_6 \Rightarrow g_7 \)
- \( x_3: g_0 \Rightarrow g_1 \Rightarrow g_5 \Rightarrow g_9 \Rightarrow g_7 \)
- \( x_4: g_0 \Rightarrow g_8 \Rightarrow g_5 \Rightarrow g_6 \Rightarrow g_7 \)
- \( x_5: g_0 \Rightarrow g_8 \Rightarrow g_5 \Rightarrow g_9 \Rightarrow g_7 \) and
- \( x_6: g_0 \Rightarrow g_8 \Rightarrow g_{10} \Rightarrow g_{11} \Rightarrow g_{12} \)

Among them, the execution sequences \( x_3 \) and \( x_5 \) are correct execution sequences, whereas the sequences \( x_2, x_3, x_4, \) and \( x_5 \) are faulty ones (they deadlock at \( g_7 \)).

However, reachability analysis suffers from the state explosion problem because the size of the RG increases exponentially with the complexity of protocol. As a result, reachability analysis and its variations are unsuitable for analyzing complex protocols.

2.2. The path-based approach

Inspecting 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, as 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 rather than the whole RG. The memory requirement to store 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.

An execution sequence can be classified as terminated, non-terminated and infinite. First of all, a terminated execution sequence is a finite one ranging from the global initial state to the global state in which every module reaches its corresponding terminal state, such as \( x_1 \) and \( x_6 \) 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 obtain a sequence of \( m \)'s transitions and this sequence will be a path of \( m \).

Definition 7. Within a system \( P \), a 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 \( s_{i+1} \).

If we perform such projection of an execution sequence with respect to every module, we can get a set of path each of which belongs to a distinct module. For example,
sequence \( x_i \) in Fig. 2, if we project the transitions of \( x_i \), 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 \( a_4 = t_{21} \).

Second, a non-terminated execution sequence is a finite sequence with at least one module not reaching its terminal state. If we perform the projection of such execution sequence, we obtain the sequence of subpaths, and the behavior of the non-terminated execution sequence can be represented by this set of subpaths.

**Definition 8.** An subpath (or infix) \( u_a \) of module \( m_a \) is defined as \( u = [s_i,t_{j+1},...,s_{j+1};t_i,t_{j+1},...,t_j] \) if its length is \( j-i+1 > 0 \) or \([s_i;]\) if it is empty, where \( 1 \leq i \leq j \leq k \).

**Definition 9.** A prefix \( x_a \) of \( p_a \) is a subpath whose heading state is the heading state of \( p_a \) and it is said to be included by \( p_a \), i.e. \( p_a = x_u \), where “·” is the subpath concatenation operator, and \( u_a \) is a subpath of \( p_a \). If \( u_a \) is empty, \( x_a \) is denoted as a pure prefix.

However, any subpath must be included by at least one path provided every state is reachable from the initial one. (This requirement should be satisfied for any correct system; otherwise, there must be 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. Thus, the behavior of such non-terminated execution sequence can also be represented by a set of paths.

Third, an infinite execution sequence must imply that there exists a cyclic structure in it, 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.)

**Definition 10.** Within a system \( P \), a nonempty cyclic path \( p_a \) in module \( m_a \) is defined as \( p_a = [s_1,\ldots,(s_i,\ldots,s_j)^*,\ldots,s_{k+1};t_i,\ldots,(t_i,\ldots,t_j)^*,\ldots,t_k] \), where \( s_1 = O_{a}, s_{k+1} = Z_{a} \) and \( t_i \) is the transition from \( s_i \) to \( s_{i+1} \). The subpath \([s_i,\ldots,s_j;t_i,\ldots,t_j]\) is the loop of \( p_a \).

Therefore, we can use a set of module paths to represent the behavior of the execution sequence:

**Definition 11.** Within a system \( P \), a **concurrent path** is defined as an ordered set of paths \( \{p_1,p_2,\ldots,p_n\} \) (or denoted as \( \{p_i\}_{i=1}^{n} \) for short), where \( p_i \) is a path of \( m_i \).

The concurrent path represents 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 [12]; the execution sequence does the same execution behavior 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 [13]. In Ref. [1], we have shown that the behavior of all execution sequences within a RG can be equivalently represented by a set of concurrent paths and we can completely verify the protocol using this set.

The advantage of using the notion of concurrent path instead of that of execution sequence is three-fold: (1) all concurrent paths can be generated automatically and within low space complexity; (2) the system is separated into a set of concurrent paths, each of which is much smaller then the original system; and (3) the verification can be performed in parallel. The first advantage results from the representation of concurrent path, i.e. a set of module paths. The complete behavior of protocol is represented by the set of all concurrent paths, which are 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 obviously not all members are concurrent paths. Thus, we have to identify the concurrent paths from Cartesian product. Each member of Cartesian product is a simplified CFSM system and can be executed. If it is a concurrent path, its execution finally reaches the terminal global state or is blocked at an erroneous state that no transitions in the original system can release this blocking situation. Otherwise, it cannot be executed normally; its execution must be blocked at an intermediate global state that the execution of the complete system will not be blocked at and that is an anomaly of error. Thus, we can determine whether a member is a concurrent path via its execution within the complexity of a concurrent path rather than the whole system.

The second advantage is also obvious. The original system is now divided into a set of concurrent paths, each of which denotes 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 reachability analysis to enumerate the potential execution sequence(s). Then, these potential execution sequences can be checked for the required properties. As each concurrent path is smaller than the original system, the memory requirement of reachability analysis is also much smaller. Therefore, the state explosion problem is alleviated.

Furthermore, the conventional parallel verification algorithms such as in Ref. [14] rely on the message passing or shared memory mechanisms to build the image of the whole reachability graph. They occasionally have to exchange newly generated global states among the parallel computers to maintain the RG consistency. Thus, a great communication cost is required. Since each concurrent path is verified independently in our approach, it is naturally done in parallel. All the information to be exchanged is the verification
result of each concurrent path and the communication cost is very low.

3. The symbolic method

3.1. General ideas

One of the principal problems of the path-based approach [1] concerns the time to check whether a concurrent path candidate is valid or not, and the bottleneck of this approach’s performance depends on the checking method. In Ref. [1], we use reachability analysis upon candidates, but the analysis is time-consuming. Thus, in this section, we propose a more efficient candidate checking method rather than the reachability analysis.

Given a concurrent path candidate \( \{ p_i \}_{i=1}^n \), we can classify it into the following categories: valid and invalid. A candidate is said to be valid, if there is at least one execution sequence corresponding to it; otherwise, it is invalid. The valid can be further classified as correct and erroneous. If its corresponding execution sequences contain errors (such as deadlocks, livelocks), it is erroneous; otherwise, it is correct. We can identify the category of a candidate as follows.

3.1.1. Valid (correct)

A candidate is correct means that all transitions must be executed successfully; that is, all messages are sent and received correctly, and the execution has no infinite loop so that the transitions after the loop cannot be executed. Thus, a candidate is correct if and only if the following three conditions are satisfied:

1. match of sending;
2. match of receiving; and
3. no infinite loop.

The first condition says that if a message is to be received successfully, it must be sent first; that is, the type of message and the order of sending and receiving must be matched. Thus, to check such a condition, we can separate the path \( p_i \) into a sending sequence \( a_{ij} \) and a receiving sequence \( b_{ji} \), and check the equivalence of \( a_{ij} \) and \( b_{ji} \) for every \( i \) and \( j \), where \( a_{ij}/b_{ji} \) is the sending/receiving message sequence of \( p_i \) with respect to \( p_j \) and is defined as follows.

**Definition 12.** Given a concurrent path candidate \( \{ p_i \}_{i=1}^n \) of the system \( P \), for every path \( p_i \) in module \( m_i \), the sequence of the sending/receiving messages is \( a_{ij}/b_{ji} \) with respect to module \( m_j \), where \( a_{ij} = b_{ji} \) if and only if the projection function \( \mu/\nu \), being defined as

\[
\mu(p_i, j) = \begin{cases} 
(\lambda(t_i), \mu(p_i', j)) & \text{if } \lambda(t_i) \in M_j \\
\mu(p_i', j) & \text{otherwise}
\end{cases},
\]

and

\[
\nu(p_i, j) = \begin{cases} 
(\lambda(t_i), \nu(p_i', j)) & \text{if } \lambda(t_i) \in M_j \\
\nu(p_i', j) & \text{otherwise}
\end{cases}
\]

where \( p_i = \{ t_i \} p'_i \) and \( p'_i \) is a subpath of \( p_i \).

If \( a_{ij} \) and \( b_{ji} \) are not equal, there are unmatched messages that cannot be received successfully. One such example is shown in Fig. 3(a), where \( a_{ij} = \text{“ab”} \) but \( b_{ji} = \text{“ba”} \). In our model, a receiving action can only be executed if the required message is at the head of the channel. Thus, in Fig. 3(a), the message sequence that module \( p_j \) wants to receive is “ba”, but that sent by module \( p_i \), the channel content is “ab” where “a” is at the head of the channel. Thus, the action in \( p_j \) to receive message “b", i.e. “+b” cannot be executed and \( p_j \) is blocked. In general, let \( a'_{ij} \) and \( b'_{ji} \) denote the longest prefixes of \( a_{ij} \) and \( b_{ji} \) that are equivalent. Then \( a'_{ij} \) and \( b'_{ji} \) contain the maximally possible transitions to be executed successfully, i.e. messages sent are received correctly. The transitions after \( b'_{ji} \) are impossible to
be executed since the execution will be blocked at the first transition after \( b'_{ij} \) due to incorrect sending from \( a_{ij} \). As for \( a'_{ij} \), the transitions after \( a'_{ij} \) may be possibly executed.

The second condition reflects the fact that a message cannot be received successfully before its sending and it also cannot be received unless it is at the head of the communication channel. One such example is shown in Fig. 3(b). This is a typical case of cyclic waiting, i.e. the first transition of every path is waiting for the second transition in another path to send the required message. To examine such a situation, we can construct a directed graph of dependencies as follows.

Add a vertex denoting every transition in the candidate. Add an edge for each dependent relation, denoting that a transition has to be executed after another. \(^1\) (All transitions in the same path must be executed sequentially; and a sending transition must be executed before its corresponding receiving transition.) Thus, if this graph has any cycle, there exists cyclic waiting. The dependency graph of Fig. 3(c) is shown in Fig. 3(d), and the heavy line shows there exists a cycle in the graph and it is the situation of receiving mismatch. As for the third condition, if there are infinite loops in the behavior of a concurrent path candidate, there is at least one path with a loop. (If the candidate with infinite loop behavior has only one path with a loop, the transitions in the loop must be sending transitions.)\(^2\) To simplify the loop behavior has only one path with a loop, the transitions is at least one path with a loop. (If the candidate with infinite exists a cycle in the graph and it is the situation of receiving mismatch. Therefore, in summary, if the concurrent path does not include any loop, we can identify the category of a concurrent path candidate by the following steps: first check the equivalence of \( a_{ij} \) and \( b_{ij} \), construct the dependency graph of \( a_{ij} \) and \( b_{ij} \), use a cycle detection algorithm \([15]\) to determine the last reachable global state, and use this global state to identify its category. Otherwise, we have to check if the global state before and after the loop in the path is the same. If they are, this is a valid (infinite loop) concurrent path. The basic algorithm to verify a concurrent path is given below:

\[ \text{verify_concurrent_path}(p_1, p_2, \ldots, p_n) \]

\[ \begin{align*}
\text{begin} & \quad \text{check all possible cycles in the concurrent path} \\
\text{for } i = 1 \text{ to } n & \quad \text{if } p_i \text{ has a loop and } p_i = p'_i(c_i)^{k_i}p''_i, \text{ then} \\
& \quad q_i = p'_i \\
\text{else} & \quad q_i = p_i \\
\text{entering_state} & = \text{simple_check}(q_1, q_2, \ldots, q_n) \\
\text{if } & \quad \text{simple_check determines the last global state using the method show above. It will be replaced by simple_check2 and simple_check3 method using the symbolic technique and thus omitted here.} \\
\text{if } & \quad \text{is in invalid/erroneous candidate} \\
\text{report } & \quad \text{as invalid/erroneous candidate} \\
\text{break} & \\
K_\text{set} & = \text{empty set} \\
\text{Repeat} & 
\end{align*} \]

\(^1\) Since dependent relations are transitive, they can be classified as direct or indirect. For any dependent relation, say \( t_q \) depends on \( t_p \), if there exists a transition \( t_q \) such that \( t_q \) depends on \( t_p \), this dependent relation is indirect; otherwise, it is direct. In our dependency graph, only the direct dependent relations are included.

\(^2\) If only one path has a loop and any receiving transitions is in the loop, the loop is impossible to be executed infinitely because the corresponding module has no loop and can provide only finite number of messages for that receiving transition. After all messages are consumed, the loop stops at the receiving transition.

\(^3\) The value of \( k_i \) can be any value that is smaller than the threshold of the module \( m_i \). This threshold denotes the maximum possible number of loop times in that module. It can be determined by the protocol designer or follows the recommendation of \([1]\), i.e. only one time of loop is enough to cover most cyclic behavior. It should also be noted that it is unnecessary that the values of all \( k_i \)s are the same.
for $i = 1$ to $n$

if $p_i$ has a loop then

Let $p_i = p_i' \cdot (c_i) \cdot p_i''$

for $j = 1$ to max$_k$ / max$_k$ denote the maximal possible number of loops and is specified by users /

Let $k_i = j$

if the combination $(k_1, k_2, ..., k_n)$ is not in the $K_{set}$ then

Add $(k_1, k_2, ..., k_n)$ into $K_{set}$

Let $q_i = p_i' \cdot c_i$

Let $q_i' = q_i \cdot p_i''$

break

else

$q_i = p_i$

$q_i' = q_i$

/ Obtain the global states after the cycle to determine the errors of infinite loop /

leaving_state = simple_check($q_1$, $q_2$, ..., $q_n$)

if \{ $q_1$, $q_2$, ..., $q_n$ \} is in invalid/erroneous candidate

report \{ $p_1$, $p_2$, ..., $p_n$ \} as invalid/erroneous candidate

continue

if entering_state = leaving_state then

report \{ $p_1$, $p_2$, ..., $p_n$ \} as valid(infinite_loop)

/ Check the complete concurrent path /

simple_check($q_1$, $q_2$, ..., $q_n$)

if \{ $q_1$, $q_2$, ..., $q_n$ \} is in invalid/erroneous candidate

report \{ $p_1$, $p_2$, ..., $p_n$ \} as invalid/erroneous candidate

continue

until $(k_1, k_2, ..., k_n) = (\max_k, \max_k, ..., \max_k)$

end cycle_check

3.2. Verification of two modules

Although the approach mentioned above is more efficient than the approach using reachability analysis in Ref. [1], the algorithm to detect a cycle in a directed graph is still time-consuming. To make further improvement, we provide a more efficient checking method to determine the last reachable global state. To ease the discussion, we first present the verification of only two modules in this subsection and show that of more than two in the next subsection. We first assume the concurrent path candidate to be checked is \{ $p_1$, $p_2$ \} in the case of two modules only; and let $a_{12} = \mu(p_1, 2)$, $a_{21} = \mu(p_2, 1)$, $b_{12} = \nu(p_1, 2)$, and $b_{21} = \nu(p_2, 1)$.

As described above, there are two reasons of blocking: mismatches of sending and receiving, and the former can be checked by the equivalence of $a_{12}$ and $b_{21}$; $a_{21}$ and $b_{12}$. Such check is quite easy (by the Boolean exclusive or operation) and we can use it as the first check to determine the maximal possible reached transitions. Let $a_{12}'$, $b_{21}'$, $a_{21}'$ and $b_{12}'$ be the maximal equivalent prefixes of $a_{12}$, $b_{21}$, $a_{21}$ and $b_{12}$, respectively. The transitions behind $b_{21}'$ and $b_{12}'$ are impossible to be executed due to the incorrect order of sending.

Then, with respect to $b_{21}'$ and $b_{12}'$, we have to check whether the concurrent path will be blocked due the incorrect order of receiving. Instead of using the dependency graph, we can identify the blocking due to incorrect receiving order with the help of the sequence of sending or receiving, denoted as $i/o$ sequence of $p_i$.

Definition 13. The $i/o$ sequence $io_{ij}$ of a path $p_i$ of module $m_i$ with respect to module $m_j$ is defined by the function $o$, i.e. $io_{ij} = o(p_i, j)$, with $o(p_i, j)$ being defined as

\[
o(p_i, j) = \begin{cases} (+, o(p_i', j)) & \text{if } t_i \text{ is a receiving transition w.r.t module } m_j \\
(-, o(p_i', j)) & \text{if } t_i \text{ is a receiving transition w.r.t module } m_j
\end{cases}
\]

where $p_i = \{ t_i \} \cdot p_i'$ and $t_i$ is a transition and $p_i'$ a subpath of $p_i$.

With these two $i/o$ sequences $io_{12} = o(p_1, 2)$ and $io_{21} = o(p_2, 1)$ ($p_1/p_2$ are the subpaths of $p_1/p_2$ corresponding to $b_{12}'$ and $b_{21}'$), we can use two counters $q_1$ and $q_2$ to denote the number of messages in the channel to determine whether a transition can be executed. Since the order of sending and receiving are matched with respect to $p_1$ and $p_2$, a message sent to the channel will be received successfully. If the value of $q_1$ or $q_2$ is zero, it means the channel is currently empty and no receiving is possible. When both counters have the values of zeros and the next transitions to be executed are both receiving transitions, then a blocking situation occurs and the last reachable global state can be identified accordingly.

We can perform such check using the algorithm below: At first $q_1$ and $q_2$ are reset to zeros and $io_{12}$ and $io_{21}$ are examined sequentially. When the inspection reaches a “-” in $io_{12}$ or $io_{21}$, $q_1$ or $q_2$ are, respectively, increased by one; when a “+” is regarded in $io_{12}$ or $io_{21}$, $q_2$ or $q_1$ are, respectively, decreased by one. Provided the value of $q_2$ or $q_1$ does not become negative. If it would become negative, check the other sequence to see whether it can be further advanced to next transition. If it cannot (due to the fact that its counter would become negative as well), it means that both are blocked at the receiving transitions that will make
its counter negative. This algorithm is formally described as follows:

\[ \text{simple\_check2}(p_1, p_2) \]

\[ \begin{align*}
  a_{ij} &= \text{the sending message list of } p_1 \text{ with respect to } p_j \\
  b_{ij} &= \text{the receiving message list of } p_i \text{ with respect to } p_j \\
  l' &= \text{The following two statements compute sending match} \\
  \xor_1 &= \text{The position of the element in } p_1 \text{ corresponding to the first occurrence of non-zero value in } a_{21} \oplus b_{12} \\
  l' &= \text{Compute the first non-identical element in } a_{21} \text{ and } b_{12} \\
  \xor_2 &= \text{The position of the element in } p_2 \text{ corresponding to the first occurrence of non-zero value in } a_{12} \oplus b_{21} \\
  l' &= \text{Compute the first non-identical element in } a_{12} \text{ and } b_{21} \\
  io_1 &= o(\bar{p}_1, 2), \bar{p}_1 \text{ the prefix of } p_1 \text{ with the length of } \xor_1 \\
  io_2 &= o(\bar{p}_1, 1), \bar{p}_2 \text{ the prefix of } p_2 \text{ with the length of } \xor_2 \\
  q_1 &= 0; l' \text{ The number of messages in the channel from } p_1 \text{ to } p_2 \\
  q_2 &= 0; l' \text{ The number of messages in the channel from } p_2 \text{ to } p_1 \\
  x_1 &= 0; l' \text{ Next transition in } p_1 \text{ to be inspected} \\
  x_2 &= 0; l' \text{ Next transition in } p_2 \text{ to be inspected} \\
  l' &= \text{compute the receiving match} \\
  \text{Repeat} \\
  \text{progress} &= \text{False} \\
  \text{if } (io_1[x_1] = "-") \\
  l' &= \text{current transition is sending transition} \\
  x_1^+ &= \text{current transition is sending transition} \\
  q_1^+ &= \text{progress} = \text{True} \\
  \text{else if } q_2 > 0 \text{ then} \\
  l' &= \text{current transition is receiving transition and the message channel is not empty} \\
  x_2^- &= \text{current transition is sending transition} \\
  q_2^- &= \text{progress} = \text{True} \\
  \text{if } (io_2[x_2] = "-") \\
  l' &= \text{current transition is sending transition} \\
  x_2^+ &= \text{current transition is sending transition} \\
  q_2^+ &= \text{progress} = \text{True} \\
  \text{else if } q_1 > 0 \text{ then} \\
  l' &= \text{current transition is receiving transition and the message channel is not empty} \\
  x_1^- &= \text{current transition is receiving transition} \\
  q_1^- &= \text{progress} = \text{True} \\
  \text{until } \text{progress} = \text{False} \text{ l' no more transition to be executed} \\
  l' &= \text{identify the last global state} \\
  \text{for } i = 1 \text{ to } 2 \\
  \text{for } j = 1 \text{ to } 2 \\
  c_{ij} &= \text{revert}(a_{ij} - b_{ij}) \\
  l' &= \text{revert} \text{ converts the string to another in the reverse order} \backslash l' \text{ "-" is the quotient operator on strings and} \backslash l' a_{ij} - b_{ij} \text{ returns a substring whose element} \\
  \text{return} \text{ the global state } (s_1, s_2, c_{11}, c_{12}, c_{21}, c_{22}) \\
  \text{end simple\_check2} \\
\]

3.3. Verification of more than two

In Section 3.2, we explained how to use simple Boolean and arithmetic operations to compute the blocking transitions and global states in the case of two modules. The similar concept can be applied to the case of more than two modules using the sending, receiving and i/o sequences of paths. However, we have to extend the variables used in the algorithm simple\_check2 as follows:

The main extension concerns the i/o sequence, which now, for the case of \( p_i \), becomes a set of i/o sequences, each of which, say \( io_{ij} \), corresponds to communication between two modules \( m_i \) and \( m_j \). Then, \( io_{ij} = o(\bar{p}_j, j) \), where \( \bar{p}_i \) is a prefix of \( p_i \), corresponds to the shortest \( b_{ij}^{l} \) for all \( j \), and \( a_{ij}^{l} \) and \( b_{ij}^{l} \) denote the longest prefixes which are common in \( a_{ij} \) and \( b_{ij} \). Since the i/o sequence of a path is now separated into a set of i/o sequences, we cannot know the original position of each sending and receiving action in the path, and we have to record the original position in the path \( p_i \) for each i/o action. Thus, the i/o sequence for the concurrent path candidate becomes \( (act_{io}^{pos})_{l=1} \), where \( act_{io} \) is "-" or "+" denoting a sending or receiving transition, \( pos_{io} \) denotes the original position of this action in the path, and \( l \) is the number of elements in the sequence. For example, the i/o sequences for the paths in Fig. 4 are \( io_{12} = (-3, +1) \) and \( io_{13} = (+2, -4) \) for \( p_1 \), \( io_{23} = (+1, -2) \) and \( io_{23} = (-3, +4) \) for \( p_2 \), and \( io_{31} = (-2, +3) \) and \( io_{32} = (+4, -4) \) for \( p_3 \).

In addition, since there are different communication channels between a module and its neighbors, each channel between module \( m_i \) and \( m_j \) corresponds to a counter \( q_{ij} \) to record the number of messages in the channel. Two types of additional counter are also required, the counter \( x_i \) of path \( p_i \) to identify the next transition to be inspected and \( e_{ij} \) of the i/o sequence \( io_{ij} \) to record the next element to be inspected.

Then for each action in the i/o sequence, if its original position is identical to the counter \( q_i \), it means this action should be executed at this moment. If it is a sending action; or receiving action and the corresponding channel counter is larger than zero, it can be executed and the related counters
are updated accordingly. If the result of the last step shows the path counter exceeds the length of the path, all transitions in this candidate can be executed; otherwise, there is a blocking situation. We show inspection steps of a non-blocking (as shown in Fig. 4) and blocking examples (as shown in Fig. 3(c)) in Fig. 5(a) and (b), respectively.

In the example of Fig. 5(a), the result of the last step (step 13) showing that the path counter exceeds the length of the path and the fact that every channel is zero, identify this candidate as correct. In that of Fig. 5(b), if there are transitions that cannot be executed, the execution blocks at these transitions. The detailed algorithm is as follows:

\[
\text{simple-check3}(p_1, p_2, \ldots, p_n)
\]

\[
\{ \\
\quad a_{ij} = \text{the sending message list of } p_i \text{ with respect to } p_j \\
\quad b_{ij} = \text{the receiving message list of } p_i \text{ with respect to } p_j \\
\quad \text{xor}_{ij} = \text{the position of the element in } p_i \text{ corresponding to the first occurrence of non-zero value in } a_{ji} \oplus b_{ij} \\
\quad l^* \text{ compute the sending match } "l" \\
\quad y_i = \min(\prod_{j=1}^{n} \text{xor}_{ij}) \\
\quad io_{ij} = i/o \text{ sequence of } p_i \text{ with respect to } p_j \text{ before the } (y_i + 1)\text{th transition} \\
\quad l^* \text{ } io_{ij}(i)\cdot p = \text{the } p\text{th transition in the original path } "l" \\
\quad l^* \text{ } io_{ij}(i)\cdot x = "-" \text{ or } "+" \text{ denoting a sending or receiving action } "l" \\
\quad e_{ij} = 0 \quad l^* \text{ the current position in } io_{ij} "l" \\
\quad q_{ij} = 0 \quad l^* \text{ the message number of the channel } p_j \rightarrow p_i \text{ } "l" \\
\quad x_i = 0 \quad l^* \text{ the next transition to be executed in } p_i \text{ } "l" \\
\quad \text{progress} = \text{True } l^* \text{ flag to exit the check } "l" \\
\quad l^* \text{ compute the receiving match } "l" \\
\quad \text{Repeat}
\]

---

**Fig. 4.** Non-blocking example of concurrent path candidate.

**Fig. 5.** Inspection by i/o sequences.
For $i = 1$ to $n$

For $j = 1$ to $n$

progress = false

if $io_{ij}(e_{ij})p = x_i$ then

the corresponding transition of $io_{ij}(e_{ij})$ should be executed $'$

if $io_{ij}(e_{ij})x = "-"$ then $'$ a sending transition $'$

$e_{ij}^{++}$ $'$ increase the counter of $io_{ij}$ $'$

$q_{ij}^{--}$ $'$ increase no. of msg in channel $p_i \rightarrow p_j$ $'$

$x_i^{++}$ $'$ increase the counter of $p_i$ $'$

progress = true

else $'$ $io_{ij}(e_{ij})x = "+"$ a receiving transition $'$

if $x_i(j) > 0$ then

$e_{ij}^{++}$ $'$ increase the counter of $io_{ij}$ $'$

$q_{ij}^{--}$ $'$ decrease no. of msg in channel $p_i \rightarrow p_j$ $'$

$x_i^{++}$ $'$ increase the counter of $p_i$ $'$

progress = true

Until progress = False $'$ no more transition to be executed $'$

$'$ identify the last global state $'$

for $i = 1$ to $n$

$s_i = \alpha$(the $x$th transition in $p_i$)

for $i = 1$ to $n$

for $j = 1$ to $n$

$c_{ij} = revert(a_{ij} - b_{ij})$ $'$ revert converts the string to another in the reverse order $'$

end simple_check3

4. Conclusion

The “state explosion problem” in protocol verification raises two issues: large memory requirement and long verification time. For the former issue, we have proposed the path-based approach to separate the protocols into a set of concurrent paths. Each one can be generated and verified independently of the others [1]. Thus, the memory to store reachable global states depends on the complexity of a concurrent path rather than the whole protocol, and the memory space issue is alleviated.

As for the later issue, in this paper, we show a performance improvement technique of the path-based approach to compute the last reachable global state of every concurrent path candidate more efficiently. Both algorithms (simple_check2 and simple_check3) linearly check every transitions in the paths with the time complexity of $O(n^2 \times l)^2$, where $n$ is the number of modules and $l$ is the average of a path. For each iteration of check, only the exclusive-or, increment, and decrement operations are required. Therefore, the long verification time issue is also alleviated.

Although the checking method shown in this paper is efficient in identifying the concurrent paths from the Cartesian product of the module paths, the main limitation results from the model. Since the channel between two modules is independent of the others in the underlying CFSM model, the $io$ sequences of paths can thus be inspected independently of the others. There is the situation that the incoming channels from different modules share a common queue [16]. In this case, the receiving of a message does not only depend on the match of sending and receiving of the corresponding module, but also the execution speeds of all the modules that may send messages at this moment. When a receiving transition is waiting to receive a message from a module but another module executes faster and sends a message in advance, it occupies the head of the queue, and this transition cannot be executed successfully. Thus, our algorithm to determine whether a message can be received must check every possible combination in the contents of common queue resulting from different execution speeds of modules.

References


