HKU CS Tech Report TR-2012-01

Postprint of article in *Information Sciences* **194** (7): 254–269 (2012)

An enhanced flow analysis technique for detecting unreachability faults in concurrent systems☆,☆☆

Tsong Yueh Chen[a], Peifeng Hu[b], Hao Li[c], T.H. Tse[c]

[a] *Centre for Software Analysis and Testing, Swinburne University of Technology, Hawthorn 3122, Australia*
[b] *China Merchants Bank, 21/F, 12 Harcourt Road, Central, Hong Kong*
[c] *Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong*

Abstract

We present a flow analysis technique for detecting unreachable states and actions in concurrent systems. It is an enhancement of the approach by Cheung and Kramer. Each process of a concurrent system is modeled as a finite state machine, whose states represent process execution states and whose transitions are labeled by actions. We construct dependency sets incrementally and eliminate spurious paths by checking the execution sequences of actions. We prove mathematically that our algorithm can detect more unreachability faults than the well-known Reif/Smolka and Cheung/Kramer algorithms. The algorithm is easy to manage and its complexity is still polynomial to the system size. Case studies on two commonly used communication protocols show that the technique is effective.

Key words: concurrency; distributed systems; reachability analysis; static analysis

1. Introduction

Concurrent systems such as network protocols and gate-level hardware are more complex than conventional sequential programs. Communications among processes in such systems may bring about synchronization anomalies such as deadlock or starvation. Reliable communications is of prime importance [36]. It is essential to provide an effective and tractable technique for detecting concurrency faults [22].

Reachability analysis is a static method that analyzes the reachability properties of nodes and edges in the flow graph model that represents program behavior. Reachability analysis has been widely used to detect concurrency faults. According to the IEEE Standard Glossary of Software Engineering Terminology [18], a *fault* is an "incorrect step, process, or data definition". A concurrent system may be modeled, for example, as a set of communicating finite state machines with synchronous communications. Thus, the presence of any state or action unreachable from the initial state of the finite state model indicates the presence of a fault, which will be referred to as an *unreachability fault*. In particular, unreachability faults in concurrent systems are mainly caused by anomalies in synchronization. Unfortunately, because of the state explosion problem, it is infeasible to search exhaustively for unreachable states or actions in concurrent systems: the number of states increases exponentially with the number of processes [27]. Many

☆ © 2011 Elsevier. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from Elsevier.

☆☆ This research is supported in part by a grant of the Australian Research Council and a grant of the Research Grants Council of Hong Kong (Project No. 717308).

Corresponding author. Tel.: +852 2859 2183; fax: +852 2858 4141. E-mail address: thtse@cs.hku.hk (T.H. Tse).

techniques have been proposed to resolve the problem in the literature. They include flow analysis [7, 8, 10, 12, 13, 19, 26, 28], compositional analysis [1, 5, 9, 21, 31, 35], and model checking [2, 4, 14, 15, 17, 23, 24]. In particular, flow analysis with reachability graph techniques are recognized as the most precise and popular [13, 25].

In this paper, we present an approximate flow analysis technique for detecting unreachable states and actions in concurrent systems. It improves on an existing technique by Cheung and Kramer [6, 8]. Like most flow analysis techniques, the enhanced algorithm strikes a balance between accuracy and computational complexity. It can detect provably more unreachability faults in concurrent systems than current algorithms, and is polynomial in complexity to the system size. To the best of our knowledge, this is the first paper that identifies and solves the shortcomings of the well-known Reif/Smolka and Cheung/Kramer algorithms in revealing unreachability in concurrent systems. It is also the first work that applies the reachability algorithms to case studies on two commonly used protocols — the alternating bit protocol and the CSMA/CD protocol. The analyses show that a more effective algorithm is indeed required for solving the problems in the applicability of the Cheung/Kramer algorithms to popular communication protocols. The proposed algorithm bridges this important real-world gap.

The paper is organized as follows: Section 2 reviews different concurrency fault detection techniques. Section 3 defines the flow graph model of concurrent systems used in this paper. In Section 4, we analyze the algorithms by Cheung and Kramer. In Section 5, we propose a new algorithm. Section 6 analyzes the effectiveness and complexity of our algorithm. Section 7 further studies the applicability of the algorithm to real-life communication protocols. Finally, the conclusion is presented in Section 8.

2. Related work

Various static techniques have been proposed for detecting concurrency faults. They can be classified into three categories:

(a) Flow Analysis

Flow analysis is a static approach to analyze the properties of a program based on the flow graph model that represents the system behavior. Because of the need to strike a balance between precision and efficiency, most flow analysis techniques are approximate methods. The use of flow graphs was first presented by Taylor [34]. The program flow graph, annotated with synchronization constraints, was used to generate a state-transition graph representing the concurrency history. The technique enumerated all the possible execution paths based on concurrency histories for the analysis of programs having rendezvous-like synchronization. Peng and Puroshothaman [26] set up flow equations to compute the set of pending messages in the queues at any given state of system. They proposed an approximate solution for the ensuing equations. Duesterwald et al. [12] developed a framework for generating algorithms for demand-driven data flow analysis. It was based on a partial search that proceeds in the reverse direction of an exhaustive data flow analysis. Reif and Smolka [28] considered the problem of reachability in a concurrent system that communicates over unbounded buffers. They presented an approximate linear algorithm for detecting unreachable statements in programs using Petri nets. Cheung and Kramer [6, 7, 8] refined the work of Reif and Smolka. They used dependency sets and history sets to eliminate many of the spurious paths and obtain better results. Cobleigh and others [10, 13] presented FLAVERS, a finite state verification approach that analyzed the behavioral properties of concurrent systems. They proposed to express such behavioral properties in terms of event patterns. A flow graph would then be generated automatically by FLAVERS. In this way, event sequences, synchronization, and message passing could be modeled without the need to traverse the state space. They pointed out [13], however, that their static model is less precise than reachability graph approaches such as that by Cheung and Kramer. Iyer and Ramesh [19] presented an apportioning technique to tackle safeness and efficiency problems in concurrent object-oriented programs. A program synchronization point was said to be local if it interacted with another method of the same object; otherwise, it was known as global. An abstract representation of every class was produced by removing its global points, while an abstract representation of the whole program was produced by removing all the local points. Subproperties at each synchronization point were checked via the respective reachability graphs generated. However, the checking of reachability subproperties relied on existing standard algorithms.

(b) Compositional Techniques

Generally, compositional techniques are based on a global compositional model that represents system behavior. Compositional techniques are considered more suitable for analyzing systems with well-defined subsystems. They exploit modularity by dividing the system into smaller subsystems. They then verify each subsystem individually and combine the results incrementally. Context constraints [9] and incremental integration [21] have been used to simplify the compositional procedure. Tsai and Juan [35] developed heuristic techniques for efficient compositional verification of component-based software systems. Sampaio et al. [31] applied the notion of refinement checking in communicating sequential processes (CSP) [29] to compositional verification. Ahrendt and Dylla [1] presented a system for the compositional verification of Creol, an object-oriented model for distributed concurrent applications [20].

(c) Model Checking

Model checking refers to the techniques for verifying the conformance of behavior between an encoded finite-state transition system and the specification, which is usually written as formulas in temporal logic. Binary decision diagrams (BDDs) [4, 23] were first used to encode a transition relation symbolically. A propositional temporal logic formula was encoded in a BDD to represent graph transition rules. Rather than encoding all states, Enders et al. [14] improved the technique by using BDDs to encode the next-state function. Holzmann [17] devised a SPIN model checker for verifying program properties. A concurrent system was described by the process metalanguage PROMELA. Properties to be verified were expressed as linear temporal logic formulas. Anderson et al. [2] demonstrated how model checking could be used to verify the correctness of e-commerce protocols. Francesco et al. [15] proposed a user-friendly interface for model checking of system properties expressed in temporal logic. Meolic et al. [24] proposed a new temporal logic with an additional *unless* operator for model checking.

However, although symbolic encoding can reduce the space and time required for model checking, it proves to be very difficult to analyze the effectiveness of such techniques.

Among these approaches, flow analysis is the most popular [25] because it is conceptually simple and relatively straightforward to automate. It has been adopted in various applications [33]. In particular, reachability graph techniques have been recognized as more precise than other techniques in flow analysis [13]. The algorithm in this paper falls under this category. It is an enhancement of the algorithms by Cheung and Kramer [6, 8]. It detects unreachable states and actions in a concurrent system that communicates through unbounded buffers.

3. Flow graph model

Following Cheung and Kramer [7, 8], we will use finite state machines (FSMs) as the flow graph model of concurrent systems in our analysis. Each process in a concurrent system is modeled as a communicating finite state machine. The nodes in the flow graph represent the execution states of the process while the arcs represent the transitions labeled by actions that may need to be synchronized across different processes. Formally, the FSM of a process P is a quadruple $\langle State, \Sigma, \rightarrow, P_0 \rangle$ such that

(i) $State$ is a set of states.

(ii) Σ is a set of actions, known collectively as the *alphabet* of P.

(iii) $\rightarrow \subseteq State \times \Sigma \times State$ is a relation among an action and two states. For the sake of readability, we will write $(S_1, a, S_2) \in \rightarrow$ as $S_1 \xrightarrow{a} S_2$ for any $a \in \Sigma$ and any $S_1, S_2 \in State$. Each $S_1 \xrightarrow{a} S_2$ is known as a *transition*.

(iv) P_0 is the *initial state*.

Although the notation is slightly different from that of Cheung and Kramer [6], we adopt the same philosophy for the synchronization of processes: A transmitting action in one process and a receiving action in another process are labeled by the same alphabet. They can only be executed synchronously. We will refer to them as synchronous actions. On the other hand, every internal action is labeled by a unique alphabet and does not need to be synchronized with any other action.

process A
x, y, z, w: variables;
loop
  read(x, y, z);
  0: send x to channel a;
  1: send y to channel b;
  2: send z to channel c;
  3: receive w
    from channel d;
  print(w);
end loop;
	process B
	p, q, r: variables;
	loop
	  read(p);
	  0: receive q
	    from channel a;
	  if $p < q$ then
	    1: receive r
	      from channel c;
	  else
	    1: send p to channel d;
	    2: receive r
	      from channel e;
	  end if;
	  print(r);
	end loop;
		process C
		s, t: variables;
		loop
		  read(s);
		  0: receive t
		    from channel b;
		  1: send $s+t$
		    to channel e;
		end loop;

Figure 1: A system $\mathcal{S}$ of 3 processes A, B, and C.

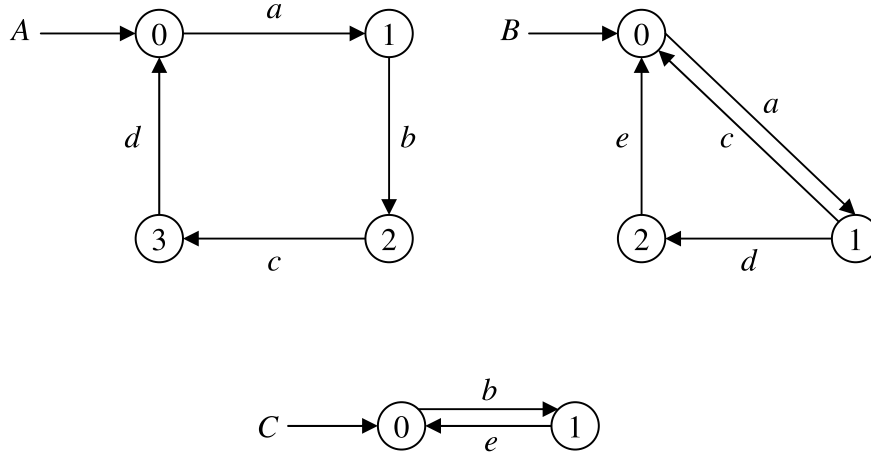


Figure 2: FSMs of processes A, B, and C.

We note also that no restriction is imposed on the relation "$\rightarrow$" in (iii). In other words, nondeterminism is allowed in FSMs at the same level of generality as Cheung and Kramer [6, 8].

Consider, for example, a system consisting of three processes A, B, and C as shown in Fig. 1, where a, b, c, d, and e are synchronous actions. The FSMs that model their behavior are shown in Fig. 2.[1] For process A, for instance,

$$\begin{aligned} State &= \{A_0, A_1, A_2, A_3\} \\ \Sigma &= \{a, b, c, d\} \\ \rightarrow &= \{A_0 \xrightarrow{a} A_1, A_1 \xrightarrow{b} A_2, A_2 \xrightarrow{c} A_3, A_3 \xrightarrow{d} A_0\} \\ P_0 &= A_0 \end{aligned}$$

Unreachability in concurrent systems is different from that in sequential programs. In sequential programs, unreachable statements are mainly due to problematic logic in certain paths. In concurrent systems, however,

[1]In the FSMs, we will ignore all the internal actions and only consider the states involving synchronous actions. Please refer to Section 5.2 for more details.

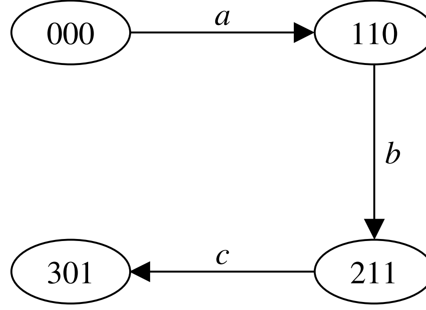


Figure 3: Global FSM of system $\mathcal{S}$.

unreachable states and actions may be due to synchronization anomalies. In our analysis, we only consider the unreachability caused by synchronization anomalies, since we can use a conventional path analysis technique to detect unreachable statements due to problematic logic.

We introduce the concept of a global FSM to explain the reachability and unreachability in concurrent systems. The *global* FSM of a concurrent system is a finite state machine that represents the behavior of the whole system. It can be constructed by applying the composition operator $||$, which is similar to the interaction operator in CSP [29]. Given $T_1 = \langle State_1, \Sigma_1, \rightarrow_1, P_1 \rangle$ and $T_2 = \langle State_2, \Sigma_2, \rightarrow_2, P_2 \rangle$, the composition FSM of $T_1 \mid\mid T_2$ is defined as the tuple $\langle State, \Sigma, \rightarrow, P_0 \rangle$ such that

$$\begin{aligned} State &= State_1 \times State_2 \\ \Sigma &= \Sigma_1 \cup \Sigma_2 \\ P_0 &= (P_1, P_2) \end{aligned}$$

and "$\rightarrow$" is given by the following transition rules:

$$\frac{P \xrightarrow{a}_1 P'}{(P,Q) \xrightarrow{a} (P',Q)} \text{ if } a \notin \Sigma_2$$

$$\frac{Q \xrightarrow{a}_2 Q'}{(P,Q) \xrightarrow{a} (P,Q')} \text{ if } a \notin \Sigma_1$$

$$\frac{P \xrightarrow{a}_1 P \text{ and } Q \xrightarrow{a}_2 Q'}{(P,Q) \xrightarrow{a} (P',Q')} \text{ if } a \in \Sigma_1 \cap \Sigma_2$$

$T_1 \mid\mid T_2$ is the composition FSM of T_1 and T_2. The $||$ rule states that if an action a is common to both alphabets Σ_1 and Σ_2, it must be executed synchronously by both processes.

Fig. 3 is the global FSM of the system $\mathcal{S}$ that consists of processes A, B, and C.

The global FSM of a concurrent system may have many possible execution paths. We say that a global state S or an action a is *reachable* if it is contained in some execution path leading from the initial state; otherwise, we say that it is *unreachable*. A state S_i in a process T_i is *reachable* if it is part of a reachable global state; otherwise, it is *unreachable*. An action a in a process T_i is *reachable* if it is reachable in the global FSM; otherwise, it is *unreachable*. For system $\mathcal{S}$ in Fig. 2, for instance, state B_2 and actions d and e are unreachable.

In theory, by checking the reachability properties of the global FSM, we can obtain the reachability properties of each process. Unfortunately, the number of states in a global FSM increases exponentially with the number of processes in the integrated system, thus resulting in the state explosion problem [27].

We propose a flow analysis technique for detecting unreachability faults using the flow graph model of each process without the need to construct a global FSM. It is an enhancement of the algorithms by Cheung and Kramer [6,

8]. In their algorithms, not all the necessary criteria for true reachability have been verified. Hence, an action or state identified as "reachable" by these algorithms may not be truly reachable. Fortunately, all truly reachable states or actions will not be wrongly identified as unreachable by their algorithms. Thus, all the actions or states that are not identified as reachable are truly unreachable.

Compared with existing algorithms, the new algorithm uses more stringent criteria to obtain a more precise reachable set. It can detect provably more unreachability faults in concurrent systems. The complexity of the algorithm naturally increases, but is still polynomial in relation to the system size. Please refer to Section 6.2 for more details.

4. Algorithms by Cheung and Kramer

The work of Cheung and Kramer [8] is based on the original algorithm by Reif and Smolka [28], which we will call *Algorithm A*. An outline of Algorithm A is as follows. All the states specified in the algorithm are local states without the need to construct a global FSM.

(i) Set the initial states of all the processes to be RS-reachable.[2] Set all the actions and non-initial states of all the processes to be initially unreachable.

(ii) Set an action a to be RS-reachable if each process having a in its alphabet contains a transition $S \xrightarrow{a} S'$ such that S is RS-reachable.

(iii) For the transition $S \xrightarrow{a} S'$ in (ii), set S' to be RS-reachable if a is RS-reachable. ■

The algorithm by Reif and Smolka does not consider the execution sequences of the actions in paths, so that some spurious paths that cannot be executed may be misinterpreted as reachable. As a result, some unreachable states or actions may be identified as reachable. Hence, the fault detection capability of the algorithm is limited. Cheung and Kramer [8] proposed the use of dependency sets and history sets to obtain better results.

An action b is dependent on another action a if and only if b cannot be executed unless a has been executed. A *dependency set* $\triangle_b$ is the set of actions on which b depends. A *history set* H_S is the set of actions in the reachable paths from the initial state to the current state S. The dependency sets and history sets can partially reflect the execution sequences of the actions in paths. In their algorithm, an action a is identified as CK-reachable[3] if, for any process T, there exists a transition $S \xrightarrow{a} S'$ in T such that S is CK-reachable and $\Sigma_T \cap \triangle_a$ is a subset of H_S. In other words, all the actions in the dependency set of a in T should have taken place before a is executed. As a result, more faults are detected by the constraint on action dependency. Any spurious path containing an action that does not satisfy the constraint will be eliminated.

A general description of the first algorithm by Cheung and Kramer, which we will call *Algorithm B*, is as follows:

Initialization:

(i) Set the initial states of all the processes to be CK-reachable. Set all the actions and non-initial states of all the processes to be initially unreachable. Set the history sets of all the states to be empty. For every action a, compute the dependency set $\triangle_a$. This is worked out by checking whether a can still be RS-reachable after removing another action b from all the processes. For example, after b has been removed, if action a is unreachable by Reif and Smolka's method, then $b \in \triangle_a$.

Iteration:

(ii) Set an action a to be CK-reachable if each process T having a in its alphabet contains a transition $S \xrightarrow{a} S'$ such that S is CK-reachable and $\Sigma_T \cap \triangle_a \subseteq H_S$.

(iii) For the transition $S \xrightarrow{a} S'$ in (ii), set S' to be CK-reachable if a is CK-reachable.

---

[2] Unless it is obvious from the context, we will use the expression RS-*reachable* to describe the actions and states identified to be "reachable" by Reif and Smolka's algorithm, as distinct from actions and states that are truly reachable according to the basic definition.

[3] Unless it is obvious from the context, we will use the expression CK-*reachable* to describe the actions and states identified to be "reachable" by Cheung and Kramer's algorithm, as distinct from actions and states that are truly reachable or RS-reachable.

History Set Propagation:

(iv) For any transition $S \xrightarrow{a} S'$, if S and a are CK-reachable, then set $H_{S'}$ to be $H_{S'} \cup \{a\} \cup H_S$.

Termination:

(v) The algorithm will terminate when all the history sets cease to grow. ■

Algorithm B delivers a more accurate result than Algorithm A, but does not eliminate all spurious paths either. Consider the example in Fig. 2. Action d in process A, and state B_2 and actions d and e in process B, are unreachable in the actual system. According to Algorithm B, however, action d is reachable. The following are the computations according to Algorithm B:

The dependency sets of actions are first computed using the algorithm of Reif and Smolka.

$$\begin{aligned} \triangle_a &= \{\} \\ \triangle_b &= \{a\} \\ \triangle_c &= \{a, b\} \\ \triangle_d &= \{a, b, c\} \\ \triangle_e &= \{a, b, c, d\} \end{aligned}$$

Iteration 1: States A_0, B_0, and C_0 are reachable. $H_{A_0} = \{\}$, $H_{B_0} = \{\}$, $H_{C_0} = \{\}$.

Iteration 2: States A_1 and B_1 and action a are reachable. $H_{A_1} = \{a\}$, $H_{B_1} = \{a\}$.

Iteration 3: States A_2 and C_1 and action b are reachable. $H_{A_2} = \{a, b\}$, $H_{C_1} = \{b\}$.

Iteration 4: State A_3 and action c are reachable. $H_{A_3} = \{a, b, c\}$, $H_{B_0} = \{a, c\}$, and $H_{B_1} = \{a, c\}$.

Iteration 5: Action d is reachable.

Cheung and Kramer [6] further extended the algorithm using the concept of re-reachability. An action a is re-reachable if and only if it is reachable via some execution sequence ε that contains a. A state S is re-reachable if and only if it is reachable via some execution sequence ε more than once. Their improved algorithm, which we will call *Algorithm C*, differentiates actions that can only be executed once from those that are re-reachable. They revised step (iv) of Algorithm B as follows:

History Set Propagation

(iv′) For any transition $S \xrightarrow{a} S'$, if S and a are CK-reachable, then

if a is re-reachable, set $H_{S'}$ to be $H_{S'} \cup \{a\} \cup H_S$; otherwise, set $H_{S'}$ to be $H_{S'} \cup \{a\} \cup (H_S \setminus \{b \,|\, a \in \triangle_b\})$.

As a result of this revision, H_{B_1} in iteration 4 above should be $\{a\}$ instead of $\{a, c\}$ because action a is not re-reachable. In iteration 5, action d will not be identified as reachable since $\Sigma_B \cap \triangle_d = \{a, c\} \nsubseteq \{a\} = H_{B_1}$.

Despite the introduction of the re-reachability concept, Algorithm C still has its limitations. We present a simple steam boiler system (Fig. 4) to illustrate this point. The system consists of four units: a steam boiler, a steam detector used to monitor the steam pressure in the steam boiler, a boiler controller, and a power switch to turn the power on or off. When the power is on *and* the steam pressure is low, the steam boiler will run. Otherwise, the steam boiler will stop.

The FSM in Fig. 5(a) describes the behavior of the steam detector. When a low steam pressure is detected, it notifies the boiler controller through an action *safe*. When a high pressure is detected, it notifies the controller through an action *danger*. The FSM in Fig. 5(b) specifies the behavior of the power switch. The boiler controller, modeled in Fig. 5(c), activates or de-activates the steam boiler depending on the pressure information it receives. If the steam pressure is at a safe level, as indicated by the action *safe*, it sends a *run* action to the steam boiler. If it is at a dangerous level, as indicated by the action *danger*, it sends a *stop* action to the boiler. The FSM in Fig. 6(a) shows the correct behavior of the steam boiler. When the power is on and it receives a *run* action, the steam boiler begins to run. If it

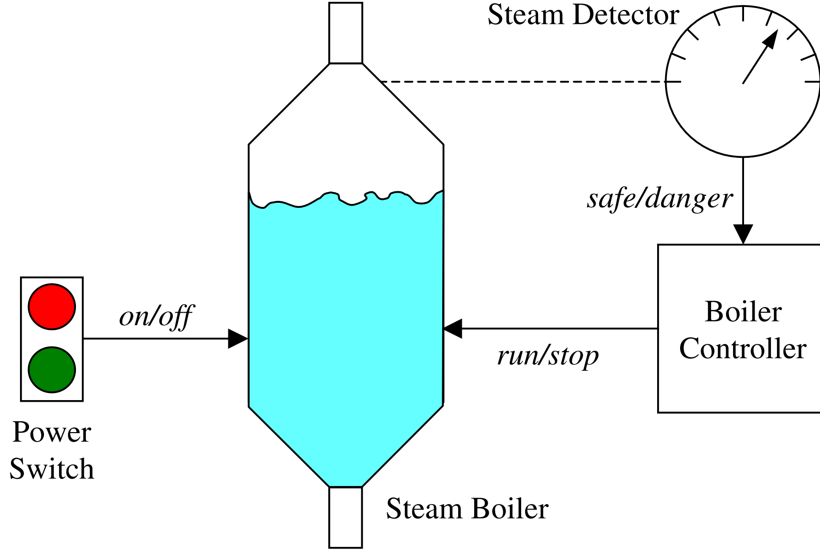


Figure 4: A steam boiler system.

receives a *stop* signal, it stops until another *run* signal is received. When the boiler is stopped, the power can be turned off, in which case the boiler returns to the initial state. On the other hand, Fig. 6(b) models the behavior of a faulty steam boiler.

Thus, the correct system is *Detector* || *Switch* || *Controller* || *Boiler*, as summarized in Fig. 7(a), while *Detector* || *Switch* || *Controller* || *Boiler′* is the faulty system, as summarized in Fig. 7(b). In the faulty system, the action *stop* is unreachable. If we use Algorithm C to detect errors in the faulty system, however, we find $\Sigma_{Controller} \cap \triangle_{stop} = \{safe, danger\} \subseteq \{safe, run, danger\} = H_{Controller_2}$ and $\Sigma_{Boiler'} \cap \triangle_{stop} = \{on\} = H_{Boiler'_1}$. Hence, Algorithm C misinterprets the action *stop* as reachable.

Algorithms A, B, and C exhibit an ascending order of capability in detecting unreachability faults in concurrent systems. The complexities of the algorithms are naturally in ascending order also. For example, given s states and n actions in a process flow graph, the complexity of Algorithm A is $O(s+n)$ while that of Algorithm C with re-reachability is $O(s \times (s+n))$ [6]. Nevertheless, even for Algorithm C, the fault detection capability is limited. First, the dependency set cannot reflect the actual dependency relationships among actions, since it is computed according to Algorithm A and, hence, inherits the limitations of the latter. Second, the history set only records all the actions in the path from the initial state to the current state. It does not record the execution sequences of all the actions in reachable paths from the initial state to the current action. The condition that "the dependency set of a belongs to the history set of S" is only necessary but not sufficient for the true reachability of a. As a result, an action satisfying the condition may still be unreachable.

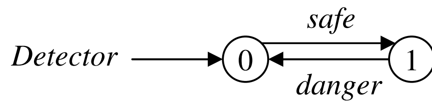
Following up on Cheung and Kramer's algorithm, we propose a new algorithm that can alleviate the limitation. We will prove in Section 6 that it can detect more unreachability faults than existing algorithms, even though it is still polynomial in complexity.

5. Our algorithm

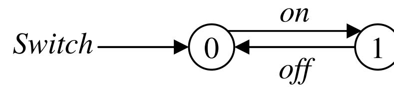
5.1. Terminology

The following is the terminology used in the proposed algorithm:

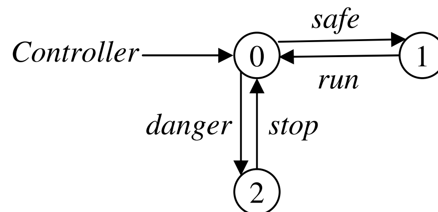
- *In-Action:* An action a is an *in-action* of state S' if and only if there exists a transition $S \xrightarrow{a} S'$.
- *Out-Action:* An action a is an *out-action* of state S if and only if there exists a transition $S \xrightarrow{a} S'$.



(a) FSM for Steam Detector

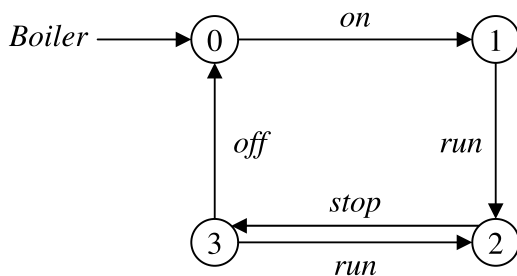


(b) FSM for Power Switch

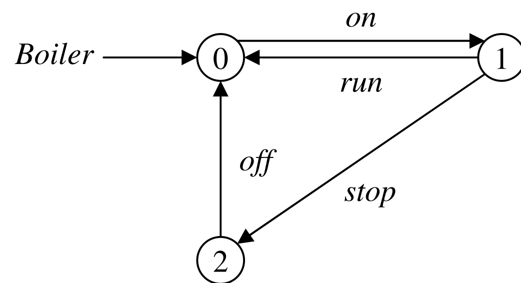


(c) FSM for Boiler Controller

Figure 5: FSMs of steam detector, power switch, and boiler controller.



(a) FSM for Correct Steam Boiler



(b) FSM for Faulty Steam Boiler

Figure 6: FSMs of correct and faulty steam boilers.

- *Predecessor:* The *predecessors* of an out-action of a state are the in-actions of that state. The default predecessor of the out-action of the initial state is the null action #.
- *Reachable Successor:* The *reachable successors* of an in-action of a state are the reachable out-actions of that state.
- *Synchronous Action:* An action a is a *synchronous action* of two processes if and only if a is in the alphabets of both processes. Intuitively, action a must be executed synchronously by these two processes.
- *Internal Action:* An action a is an *internal action* of a process if and only if a is in the alphabet of only that process. Intuitively, internal actions are executed locally without the need for synchronization. For any transition $S \xrightarrow{a} S'$, if state S is reachable and a is an internal action, then a must be reachable.

In the next section, we will show how to capture more precise information on the execution sequences of actions, with a view to eliminating spurious paths.

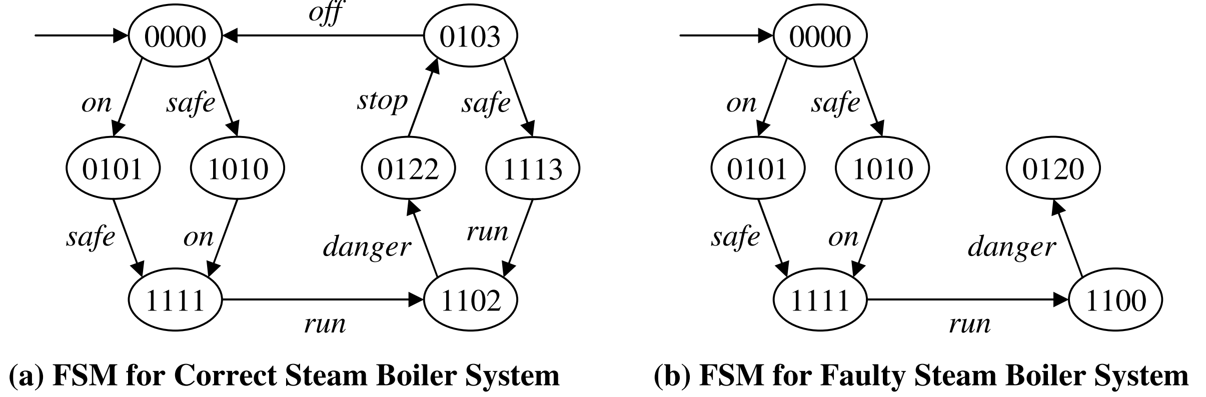


Figure 7: Global FSMs of correct and faulty steam boiler systems.

5.2. Description of algorithm

For ease of description of the algorithm, we will ignore all internal actions of each process as follows: If a process A contains a transition $S \xrightarrow{a} S'$ involving an internal action a and two states S and S', we let $S' = S$ and remove the internal action a from process A. Continuing in this way for all internal actions, the process A can be turned into a process A' that does not contain internal actions. Then, we can concentrate on the detection of unreachability faults of the simplified process A'. If state S of process A' is identified as reachable by the algorithm, then internal action a and state S' of the original process A are also identified as reachable. Otherwise, action a is treated as unreachable. Thus, in the description of the algorithm, all actions are synchronous actions in the alphabets of two or more processes. This simplification technique has been commonly used in existing algorithms.

We recall that an action a is reachable if and only if it is in some execution path of the global FSM leading from the initial state. Suppose action a is a synchronous action of processes T_1 and T_2, and it is reachable via a path ε in the global FSM. Suppose, further, that a is an out-action of a state S_1 in T_1 and an out-action of a state S_2 in T_2. Then, there must be a reachable in-action m for state S_1 in T_1 and a reachable in-action n for state S_2 in T_2. In other words, after executing m in process T_1 and executing n in process T_2, a can be executed synchronously by processes T_1 and T_2.

Similar to the history set of a state, we use H_a to denote the history set of an action a. It is the set of actions in the reachable paths from the initial state to action a in the global FSM. We define

$$\triangle_{m,n} = \triangle_m \cup \triangle_n \cup \{m, n\}$$

Since action a is reachable via the path ε, all the actions in $\triangle_{m,n}$ must be executed before a. Hence,

$$\begin{aligned} &\Sigma_{T_1} \cap \triangle_{m,n} \subseteq \Sigma_{T_1} \cap (H_m \cup \{m\}) \text{ and} \\ &\Sigma_{T_2} \cap \triangle_{m,n} \subseteq \Sigma_{T_2} \cap (H_n \cup \{n\}) \end{aligned} \tag{1}$$

In our algorithm, when we want to check whether an action a is reachable, we look for the existence of the processes T_1 and T_2 having a as a synchronous action, and check whether there is a reachable predecessor m of a in T_1 and a reachable predecessor n of a in T_2 such that statement (1) is satisfied. A state is set to be reachable if it has a reachable in-action.

A summary of the algorithm is as follows:

Initialization:

(i) Set the initial states of all the processes to be CHLT-reachable.[4] Set all the actions and non-initial states of all the processes to be initially unreachable. Set the dependency set of the null action to be empty and the history set of the null action to contain only itself.

Iterations:

(ii) For any action a that is common to any processes T_1 and T_2 and has not yet been identified as CHLT-reachable, check whether there exist a transition $S_1 \xrightarrow{a} S'_1$ in T_1 and a transition $S_2 \xrightarrow{a} S'_2$ in T_2 satisfying the following condition:

> S_1 and S_2 have been identified as CHLT-reachable and there exist a CHLT-reachable in-action m for state S_1 and a CHLT-reachable in-action n for state S_2 such that $\Sigma_{T_1} \cap \triangle_{m,n} \subseteq \Sigma_{T_1} \cap (H_m \cup \{m\})$ and $\Sigma_{T_2} \cap \triangle_{m,n} \subseteq \Sigma_{T_2} \cap (H_n \cup \{n\})$.

If so,

- set the action a to be CHLT-reachable,
- set the dependency set $\triangle_a$ to be $\triangle_{m,n}$,
- set the history set H_a to be $H_m \cup H_n \cup \{m, n\}$, and
- if the states S'_1 and S'_2 have been identified as CHLT-reachable, set them to be re-reachable; otherwise, set them to be CHLT-reachable.

For any other pair of action (m', n') $(\neq (m, n))$ that also satisfies the above condition, update the dependency set $\triangle_a$ to $\triangle_a \cap \triangle_{m',n'}$ and update the history set H_a to be $H_a \cup H_{m'} \cup H_{n'} \cup \{m', n'\}$.

Iterative Set Propagation:

(iii) For each pair of CHLT-reachable predecessors m and n of action a in processes T_1 and T_2, respectively, if H_m, H_n, $\triangle_m$, or $\triangle_n$ changes, check whether a can be reached from m and n. If it can be so reached, update the dependency set $\triangle_a$ to $\triangle_a \cap \triangle_{m,n}$ and update the history set H_a as follows:

- If the state that has m as in-action is re-reachable, set H_a to be $H_a \cup H_m$; otherwise, set H_a to be $H_a \cup (H_m \setminus \{b \,|\, m \in \triangle_b\})$.
- If the state that has n as in-action is re-reachable, set H_a to be $H_a \cup H_n$; otherwise, set H_a to be $H_a \cup (H_n \setminus \{b \,|\, n \in \triangle_b\})$.
- Set H_a to $H_a \cup \{m, n\}$.

Termination:

(v) Terminate the algorithm when no new action is identified as CHLT-reachable. ■

Compared with existing algorithms, the proposed algorithm captures information on execution sequences among actions more accurately in two ways: First, the computation of dependency sets is more precise. In our algorithm, the computation is based on the general concept that the dependency set of an action a is decided by the actions prior to a in the reachable paths of a global FSM. To avoid the state explosion problem, however, we do not want to process the entire global FSM. The trick is that we need only take into account the *two* actions immediately prior to the (synchronous) action a in the reachable paths of the global FSM. Thus, the two actions and the dependency set of these actions should be included in the dependent set of action a. When an action is identified as reachable via a path, its dependency set is computed. When the action is also found to be reachable via another path, its dependency set is updated. Second, when deciding whether an action a is reachable, the algorithm also checks how a is reached from

[4] Unless it is obvious from the context, we will use the expression CHLT-*reachable* to describe the actions and states identified to be "reachable" by our algorithm, as distinct from actions and states that are truly reachable, RS-reachable, or CK-reachable.

other actions, whereas Algorithms B and C only check whether all the other actions on which a depends are executed before a.

Let us illustrate the usefulness of the enhanced algorithm in detecting more faults through a couple of examples below. We will give a rigorous theoretical analysis in Section 6. Further case studies on real-life communication protocols will be given in Section 7.

Let us use the algorithm to check the reachability of system $\mathcal{S}$ in Fig. 2. The actions a, b, and c will be identified as reachable. Consider action c, for instance. It has a reachable predecessor b in process A and a reachable predecessor a in process B. We note that $\triangle_b = \{\#, a\}$ and $\triangle_a = \{\#\}$, so that $\triangle_{b,a} = \{\#, a, b\}$. We note also that $H_b = \{\#, a\}$ and $H_a = \{\#\}$. Hence, $\Sigma_A \cap \triangle_{b,a} = \{\#, a, b\} = \Sigma_A \cap (H_b \cup \{b\})$ and $\Sigma_B \cap \triangle_{b,a} = \{\#, a\} = \Sigma_B \cap (H_a \cup \{a\})$. On the other hand, action d will not be identified as reachable because its reachable predecessors are actions a and c but $\Sigma_B \cap \triangle_{a,c} = \{\#, a, c\} \nsubseteq \{\#, a\} = \Sigma_B \cap (H_a \cup \{a\})$.

Using the algorithm to check the reachability of the faulty steam boiler system in Fig. 6(b), the actions *on*, *safe*, *run*, and *danger* will be identified as reachable. Consider the action *run*, for example. It has a reachable predecessor *safe* in *Controller* (see Fig. 5(c)) and a reachable predecessor *on* in *Boiler*$'$ (see Fig. 6(b)). We note that $\triangle_{safe} = \{\#\}$ and $\triangle_{on} = \{\#\}$, so that $\triangle_{safe,on} = \{\#, on, safe\}$. We note also that $H_{safe} = \{\#\}$ and $H_{on} = \{\#\}$. Hence, $\Sigma_{Controller} \cap \triangle_{safe,on} = \{\#, safe\} = \Sigma_{Controller} \cap (H_{safe} \cup \{safe\})$ and $\Sigma_{Boiler'} \cap \triangle_{safe,on} = \{\#, on\} = \Sigma_{Boiler'} \cap (H_{on} \cup \{on\})$. On the other hand, the action *stop* will not be identified as reachable because its reachable predecessors are the actions *danger* and *on* but $\Sigma_{Boiler'} \cap \triangle_{danger,on} = \{\#, on, run\} \nsubseteq \{\#, on\} = \Sigma_{Boiler'} \cap (H_{on} \cup \{on\})$.

6. Algorithm analysis

6.1. Effectiveness analysis

We would like to compare the effectiveness of our algorithm, which we call *Algorithm D*, with those of Algorithms A, B, and C. Let U be the set of truly unreachable actions of a real system and U_A, U_B, U_C, and U_D be the sets of unreachable actions given by Algorithms A, B, C, and D, respectively. From Cheung and Kramer [7, 8], we have

$$U_A \subseteq U_B \subseteq U_C \subseteq U.$$

First, we prove that $U_D \subseteq U$ by proving that all the truly reachable actions in the global FSM will be identified as reachable by Algorithm D. Thus, all the actions identified as unreachable by Algorithm D are truly unreachable in the real system.

Lemma 1

Given any state G in a global FSM, if there exists a path from the initial state to G such that all the actions in the path prior to G have been identified as reachable by Algorithm D, then any out-action of G will also be identified as reachable by this algorithm.

Proof

Let $\mathcal{G} = T_1 \,||\, T_2 \,||\, \ldots \,||\, T_n$ be the global FSM, where $T_1, T_2, \ldots, T_n$ are processes. Let $G \xrightarrow{a} G'$ be a transition in $\mathcal{G}$. Suppose an out-action a is present in both T_i and T_j but absent in other processes. Suppose also that, when $\mathcal{G}$ is in state G, T_i is in state G_i and T_j is in state G_j.

Suppose G is the end state of a path ε in $\mathcal{G}$ that leads from the initial state. Let m be the in-action of state G_i in process T_i via the path ε, and n be the in-action of state G_j in process T_j via the path ε. Then, action m must be a reachable predecessor of action a in process T_i, and action n must be a reachable predecessor of action a in process T_j. Without loss of generality, suppose m is executed before n.

(i) Obviously, $\Sigma_{T_j} \cap \triangle_{m,n} \subseteq \Sigma_{T_j} \cap (H_n \cup \{n\})$ because $\triangle_{m,n} \subseteq H_n \cup \{n\}$ and $H_n \cup \{n\}$ contains at least all the actions in the path ε.

(ii) Assume $\Sigma_{T_i} \cap \triangle_{m,n} \nsubseteq \Sigma_{T_i} \cap (H_m \cup \{m\})$. In other words, there exists some (synchronous) action $x \in \Sigma_{T_i} \cap \triangle_{m,n}$ such that $x \notin H_m \cup \{m\}$. This is possible only if action x is executed between actions m and n. In other words, m is not an in-action of state G_i in process T_i via the path ε, which contradicts the definition of m. Hence, $\Sigma_{T_i} \cap \triangle_{m,n} \subseteq \Sigma_{T_i} \cap (H_m \cup \{m\})$.

Thus, the reachability criteria in statement (1) for Algorithm D are satisfied. Action a will be identified as reachable. ■

Proposition 1
All the reachable actions of a global FSM *can be identified as reachable by Algorithm D.*

Proof
According to Algorithm D, all the out-actions of the initial state of a global FSM are identified as reachable. Using induction and Lemma 1, we can conclude that all the reachable actions in a global FSM can be identified as reachable by Algorithm D. ■

Thus, we obtain $U_D \subseteq U$.

Let us now consider the relationship between U_C and U_D. We have already seen an example showing that there exists some unreachable action $x \in U_D$ such that $x \notin U_C$. We would like to prove that, in general, $U_C \subseteq U_D$.

Similarly, we can also conclude that all the states of a global FSM can be identified as reachable by Algorithm D, since any state in the global FSM can be reached from a reachable action.

Lemma 2
Given any action a, if its dependency sets computed by Algorithms D and C are $\triangle_a$ and $\triangle'_a$, respectively, then $\triangle'_a \subseteq \triangle_a$.

Proof
Assume the contrary. Then, there exists an action $x \in \triangle'_a$ such that $x \notin \triangle_a$. Since $x \in \triangle'_a$, x must be present in all the paths in the global FSM that lead from the initial state to action a. Otherwise, if x were absent from a path ε, then a could be executed via ε without executing x. According to Algorithm D, therefore, $x \in \triangle_a$, which contradicts the assumption above. Hence, we must have $\triangle'_a \subseteq \triangle_a$. ■

Lemma 3
Suppose a is an in-action of state S in process T_1 and a is identified as reachable by Algorithm D. Let H_a be the history set of action a computed by Algorithm D and H_S be the history set of state S computed by Algorithm C. Then, $\Sigma_{T_1} \cap (H_a \cup \{a\}) \subseteq \Sigma_{T_1} \cap H_S$.

Proof
Assume that $\Sigma_{T_1} \cap (H_a \cup \{a\}) \nsubseteq \Sigma_{T_1} \cap H_S$. Then, there exists an action $x \in \Sigma_{T_1} \cap (H_a \cup \{a\})$ such that $x \notin H_S$. Since $x \in H_a \cup \{a\}$, x is in a path from the initial state to action a in the global FSM. Suppose a is the in-action of a state G in the global FSM. When the global FSM is in state G, the process T_1 should be in state S. Hence, $x \in H_S$, contradicting the assumption above. We must, therefore, have $\Sigma_{T_1} \cap (H_a \cup \{a\}) \subseteq \Sigma_{T_1} \cap H_S$. ■

Proposition 2
All actions that are identified as unreachable by Algorithm C will also be identified as unreachable by Algorithm D.

Proof
Let $\triangle_x$ and $\triangle'_x$ be the dependency sets of action x computed by Algorithms D and C, respectively. Let H_x be the history set of an action x computed by Algorithm D, and H_S be the history set of a state S computed by Algorithm C. Suppose action a is the out-action of a state S_1 in process T_1 and the out-action of a state S_2 in process T_2. If action a is identified as reachable by Algorithm D, there exist a reachable in-action m of state S_1 and a reachable in-action n of state S_2 such that $\Sigma_{T_1} \cap \triangle_{m,n} \subseteq \Sigma_{T_1} \cap (H_m \cup \{m\})$ and $\Sigma_{T_2} \cap \triangle_{m,n} \subseteq \Sigma_{T_2} \cap (H_n \cup \{n\})$. Since $\triangle_a$ will be iteratively updated to either $\triangle_{m,n}$ or $\triangle_a \cap \triangle_{m,n}$, we have $\triangle_a \subseteq \triangle_{m,n}$, By Lemma 2, $\triangle'_a \subseteq \triangle_a$, so that $\triangle'_a \subseteq \triangle_{m,n}$. By Lemma 3, $\Sigma_{T_1} \cap (H_m \cup \{m\}) \subseteq \Sigma_{T_1} \cap H_{S_1}$ and $\Sigma_{T_2} \cap (H_n \cup \{n\}) \subseteq \Sigma_{T_2} \cap H_{S_2}$. Hence, $\Sigma_{T_1} \cap \triangle'_a \subseteq \Sigma_{T_1} \cap \triangle_{m,n} \subseteq \Sigma_{T_1} \cap (H_m \cup \{m\}) \subseteq \Sigma_{T_1} \cap H_{S_1}$ and $\Sigma_{T_2} \cap \triangle'_a \subseteq \Sigma_{T_2} \cap \triangle_{m,n} \subseteq \Sigma_{T_2} \cap (H_n \cup \{n\}) \subseteq \Sigma_{T_2} \cap H_{S_2}$. As a result, action a will also be identified as reachable by Algorithm C. Thus, all the actions identified as reachable by algorithm D will also be identified as reachable by Algorithm C. In other words, all the actions identified as unreachable by Algorithm C will also be identified as unreachable by Algorithm D. ■

In this way, we can conclude that $U_A \subseteq U_B \subseteq U_C \subseteq U_D \subseteq U$.

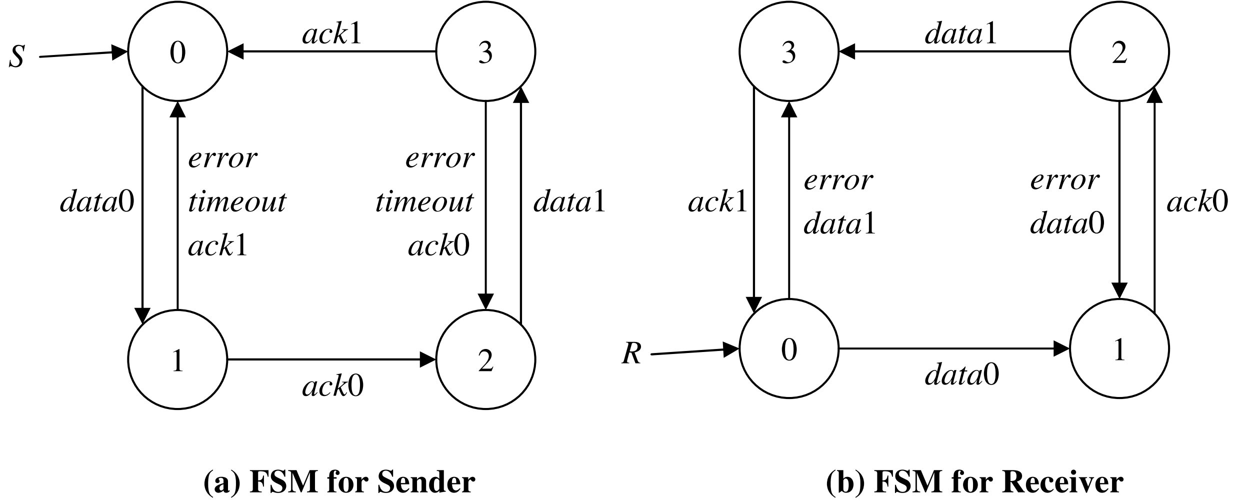


Figure 8: Original FSMs of alternating bit protocol.

6.2. Complexity analysis

In Algorithm *D*, we only store the dependency sets and the history sets. Let n be the number of actions in the system. The space complexity of the algorithm is $O(n^2+n^2)=O(n^2)$. Hence, the algorithm will not result in a state explosion problem.

Now, we discuss the time complexity of our algorithm by calculating the time complexity of each step. In step (ii), the time complexity to find a CHLT-reachable action is $O(n\times n^2)$, where $O(n^2)$ is the time complexity to decide whether an action is CHLT-reachable. We may need to check at most n actions in order to find a CHLT-reachable action. In step (iii), the time complexity of iterative set propagation is $O(n^3)$. Therefore, the total time complexity of the algorithm is $O(n\times(n^3+n^3))=O(n^4)$.

6.3. Limitation analysis

Our algorithm cannot detect all unreachability faults because, for efficiency reasons, it only checks how actions are reached from their predecessors rather than checking the execution sequences of all the actions in reachable paths, similar to other data flow methods. The criteria in the algorithm are only necessary but not sufficient conditions for true reachability. Consider, for example, Fig. 5 again. Suppose we add a transition $Detector_0 \xrightarrow{jumpstart} Detector_1$ to the FSM for steam detector in Fig. 5(a) and another transition $Boiler'_1 \xrightarrow{jumpstart} Boiler'_0$ to the FSM for the faulty steam boiler in Fig. 6(b). According to the algorithm, the action *stop* has a reachable predecessor *danger* in *Detector* and a reachable predecessor *on* in *Boiler*$'$. We note that $\triangle_{danger}=\{\#\}$ and $\triangle_{on}=\{\#\}$, so that $\triangle_{danger,on}=\{\#, on, danger\}$. We note also that $H_{danger}=\{\#, on, safe, run\}$ and $H_{on}=\{\#\}$. Hence, $\Sigma_{Detector}\cap\triangle_{danger,on}=\{\#, danger\}\subseteq\{\#, safe, danger\}=\Sigma_{Detector}\cap(H_{danger}\cup\{danger\})$ and $\Sigma_{Boiler'}\cap\triangle_{danger,on}=\{\#, on\}=\Sigma_{Boiler'}\cap(H_{on}\cup\{on\})$. As a result, the action *stop* in the modified example will be interpreted by the new algorithm as reachable. On the other hand, we realize from the global FSM of the faulty system in Fig. 7(b) that the action *stop* is, in fact, unreachable.[5]

There is no way to efficiently enumerate all execution paths because the enumeration problem is exponential in complexity. In the proposed technique, we only aim at a balance between practicality and completeness.

[5]Assume that we can add the action *stop* into the global FSM in Fig. 7(b). Since the action *danger* is a reachable predecessor, *stop* can only be added after it. Hence, there will be 3 execution paths from the initial state to the action *stop*: (i) *safe—on—run—danger—stop*, (ii) *on—safe—run—danger—stop*, and (iii) *on—jumpstart—danger—stop*. Since the actions *on*, *run*, *jumpstart*, and *stop* are in the modified *Boiler*$'$, these 3 paths contradict the fact that *on* is a predecessor of *stop*. Thus, the action *stop* cannot be added to the global FSM. In other words, *stop* is unreachable.

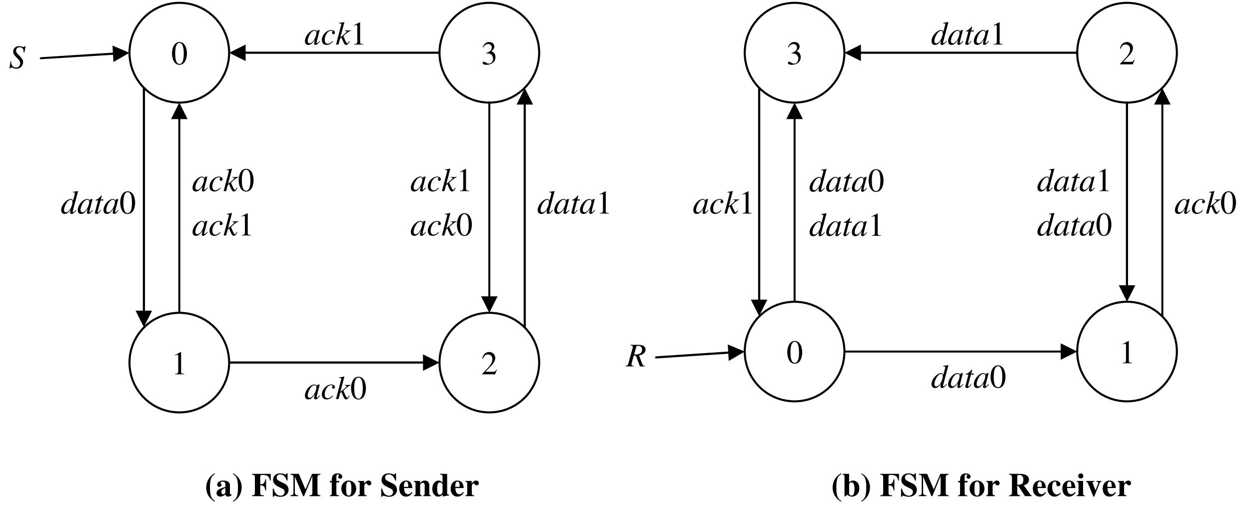


Figure 9: Nondeterministic FSMs of alternating bit protocol.

7. Further case studies

In this section, we further investigate whether our algorithm can detect real-life unreachability faults through two case studies on communicating finite state machines in common communication protocols: the alternating bit protocol [3, 11, 32] and the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol [16, 32].

Fig. 8 portrays the alternating bit protocol, which is a data-link-level communication protocol that supports the retransmission of erroneous or lost messages. The actions *data*0 and *data*1 in the sender FSM mean sending data; *data*0 and *data*1 in the receiver FSM mean receiving data; and *ack*0 and *ack*1 mean acknowledgements. Furthermore, *error* can occur in both the sender and receiver FSMs. As a result, synchronization may occur between *data*0 and *error*. Similar synchronizations may occur between *data*1 and *error*, between *error* and *ack*0, and between *error* and *ack*1. Similarly, *timeout* can occur in the sender FSM. Thus, synchronization may also occur between *timeout* and *ack*0 and between *timeout* and *ack*1. Consider a revised version of the protocol such that, for the sender FSM, *error* and *timeout* are changed to *ack*0 and *ack*1, respectively. Suppose also that, for the receiver FSM, *error* associated with the transition from state R_0 to state R_3 is changed to *data*0; *error* associated with the transition from state R_2 to state R_1 is changed to *data*1. In this revised version (as shown in Fig. 9), the state machine is nondeterministic. It is well known that, for any nondeterministic FSM, there exists an equivalent deterministic FSM. The equivalent deterministic FSMs for Fig. 9 are given in Fig. 10. Let us then consider a faulty version as shown in Fig. 11. In the sender FSM of this faulty version, the transition *ack*1 from state S_1 to state S_0 is missing, the original transition *ack*0 from state S_1 to state S_2 is changed to *ack*1, the original transition *ack*0 from state S_3 to state S_2 is changed to *data*0, and the original transition *data*1 from state S_2 to state S_3 is changed to *ack*0. In the receiver FSM, the transition *data*1 from state R_0 to R_3 is missing.

Suppose we use Algorithm *C* to check the reachability of the faulty protocol in Fig. 11. We note that $\triangle_{data0} = \{\}$, $\triangle_{ack1} = \{data0\}$, $\triangle_{ack0} = \{data0, ack1\}$, and $\triangle_{data1} = \{data0, ack1, ack0\}$. The action *data*0 will be identified as reachable, with $H_{S_1} = \{data0\}$ and $H_{R_1} = \{data0\}$. Since $\Sigma_{Sender} \cap \triangle_{ack1} = \{data0\} \subseteq \{data0\} = H_{S_1}$ and $\Sigma_{Receiver} \cap \triangle_{ack1} = \{data0\} \subseteq \{data0\} = H_{R_1}$, the action *ack*1 will be identified as reachable. Because *data*0 is re-reachable, we have $H_{S_2} = \{data0, ack1\}$, $H_{S_1} = \{data0, ack1\}$, $H_{R_0} = \{data0, ack1\}$, and $H_{R_1} = \{data0, ack1\}$. Since $\Sigma_{Sender} \cap \triangle_{ack0} = \{data0, ack1\} \subseteq \{data0, ack1\} = H_{S_2}$ and $\Sigma_{Receiver} \cap \triangle_{ack0} = \{data0, ack1\} \subseteq \{data0, ack1\} = H_{R_1}$, the action *ack*0 will also be identified as reachable. However, it is actually *unreachable*. Thus, Algorithm *C* fails to identify reachability faults in a faulty version of the classical alternating bit protocol, even though some researchers consider this protocol to be "too simple" [30].

On the other hand, we can reveal *ack*0 as unreachable using our Algorithm *D*. Since $\triangle_{data0} = \{\#\}$ and $H_{data0} = \{\#\}$, the states S_1 and R_1 are reachable. Since $\triangle_{ack1} = \{\#, data0\}$ and $H_{ack1} = \{\#, data0\}$, the action

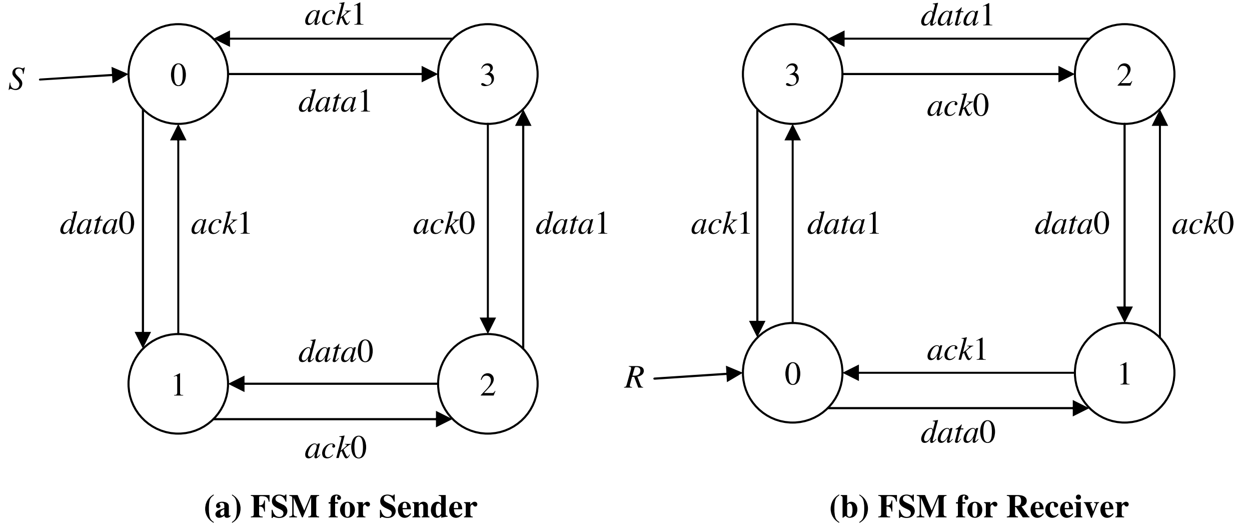


Figure 10: Deterministic FSMs of alternating bit protocol.

*ack*1 is identified as reachable. As for *ack*0, we have $\Sigma_{Receiver} \cap \triangle_{data0,ack1} = \{data0, \#, ack1\} \nsubseteq \{data0\} = \Sigma_{Receiver} \cap (H_{data0} \cup \{data0\})$. Hence, *ack*0 will be correctly identified as unreachable by the new algorithm.

Next, let us consider the CSMA/CD protocol, which has been made popular by its application to the Ethernet. Fig. 12, adapted from Gouda and Chang [16], describes the protocol. *A* and *B* are two communicating machines and *C* is the controller. Fig. 13 shows a faulty version of the protocol, where the original transitions *RqstA* from state C_0 to state C_9 and *NoRqstB* from state C_9 to state C_{12} in the controller have been swapped.

According to Algorithm *C*, $\triangle_{NoRqstA} = \{\}$, $\triangle_{NoRqstB} = \{\}$, and $\triangle_{RqstB} = \{NoRqstB\}$. *NoRqstA* and *NoRqstB* will be identified as reachable, with $H_{A_1} = \{NoRqstA\}$, $H_{B_1} = \{NoRqstB\}$, $H_{C_1} = \{NoRqstA\}$, $H_{C_2} = \{NoRqstB, NoRqstA\}$, and $H_{C_9} = \{NoRqstB\}$. Then, the action *offA* will be identified as reachable, with $H_{A_0} = \{NoRqstA, offA\}$. Similarly, *offB* will be identified as reachable, with $H_{B_0} = \{NoRqstB, offB\}$. Next, *RqstB* will also be identified as reachable because $\Sigma_B \cap \triangle_{RqstB} = \{NoRqstB\} \subseteq \{NoRqstB, offB\} = H_{B_0}$ and $\Sigma_C \cap \triangle_{RqstB} = \{NoRqstB\} \subseteq \{NoRqstB\} = H_{C_9}$. However, *RqstB* is in fact *unreachable*.

Applying the new algorithm, the actions *NoRqstA* and *NoRqstB* will be identified as reachable first, with $\triangle_{NoRqstA} = \{\#\}$, $\triangle_{NoRqstB} = \{\#\}$, $H_{NoRqstA} = \{\#\}$, and $H_{NoRqstB} = \{\#\}$. Next, *offA* will be identified as reachable, with $\triangle_{offA} = \{\#, NoRqstA, NoRqstB\}$ and $H_{offA} = \{\#, NoRqstA, NoRqstB\}$. Similarly, *offB* will be identified as reachable, with $\triangle_{offB} = \{\#, NoRqstA, NoRqstB, offA\}$ and $H_{offB} = \{\#, NoRqstA, NoRqstB, offA\}$. Let us then consider *RqstB*. Its in-actions are $\{\#, offB, NoRqstB\}$. When we take the in-actions # and *NoRqstB*, we have $\Sigma_B \cap \triangle_{\#,NoRqstB} = \{NoRqstB\} \nsubseteq \{\#\} = \Sigma_B \cap (H_{\#} \cup \{\#\})$. When we take the in-actions *offB* and *NoRqstB*, we have $\Sigma_C \cap \triangle_{offB,NoRqstB} = \{offA, NoRqstB, offB\} \nsubseteq \{NoRqstB\} = \Sigma_C \cap (H_{NoRqstB} \cup \{NoRqstB\})$. Hence, *RqstB* will be correctly identified as unreachable by our algorithm.

8. Conclusion

In this paper, we have presented a flow analysis approach to detecting unreachability faults in communicating processes. The proposed algorithm is an enhancement of the technique by Cheung and Kramer. We construct dependency sets incrementally and obtain more accurate dependency relationships by checking the execution sequences of actions. We check how an action is reached from other actions in a more precise manner. Our algorithm can detect provably more unreachability faults than the Reif/Smolka and Cheung/Kramer algorithms. It is still polynomial in complexity to number of actions. We have conducted case studies on two commonly used communication protocols. The analyses show that a more effective algorithm is indeed required for solving the

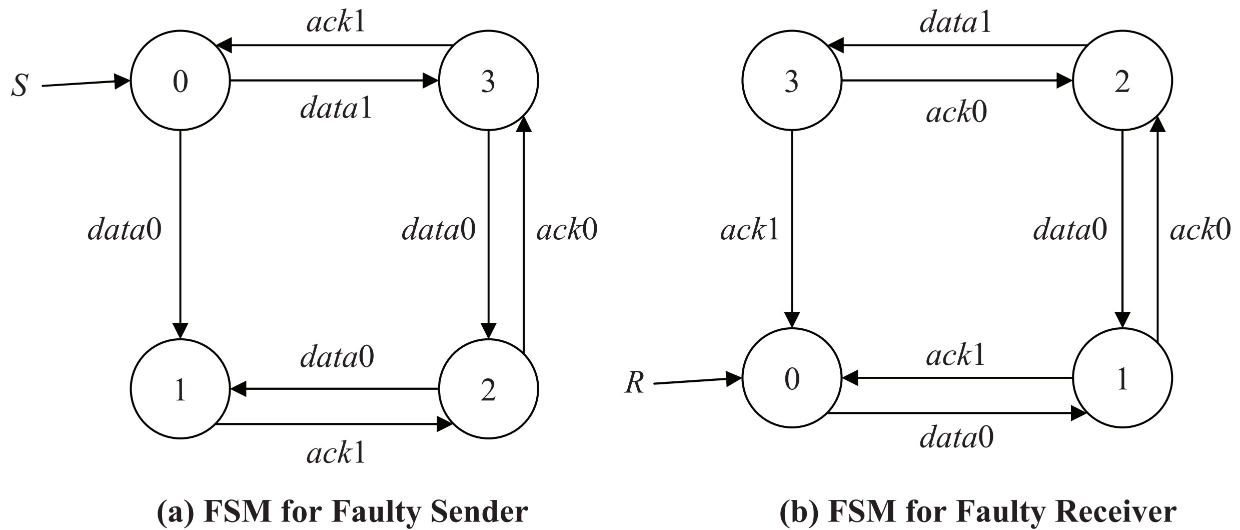


Figure 11: FSMs of faulty alternating bit protocol.

problems in the applicability of the Cheung/Kramer algorithms to real-world protocols. The enhanced algorithm bridges this important gap.

Concurrent systems are becoming extremely common. Such systems are very difficult to analyze, test, and debug because of the state explosion problem. Our approach is a good compromise between efficiency and effectiveness. It can be used for preliminary analysis, to be supplemented by other techniques in the behavioral analyses of concurrent systems, if necessary. It will be interesting future work to study how the approach may be applied in conjunction with some of the compositional or model-checking techniques for the detection of concurrency faults in Section 2.

Acknowledgments

We are grateful to Shing-Chi Cheung of the Hong Kong University of Science and Technology, Robert G. Merkel of Monash University and Dave Towey of United International College for their encouraging comments and invaluable suggestions to the paper.

References

[1] W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming (2010). doi: 10.1016/j.scico.2010.08.003.

[2] B.B. Anderson, J.V. Hansen, P.B. Lowry, S.L. Summers, Standards and verification for fair-exchange and atomicity in e-commerce transactions, Information Sciences 176 (8) (2006) 1045–1066.

[3] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, Learning communicating automata from MSCs, IEEE Transactions on Software Engineering 36 (3) (2010) 390–408.

[4] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, D.L. Dill, Symbolic model checking for sequential circuit verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13 (4) (1994) 401–424.

[5] Y.-P. Cheng, M. Young, C.-L. Huang, C.-Y. Pan, Towards scalable compositional analysis by refactoring design models, in: Proceedings of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC 2003/FSE-11), ACM, New York, NY, 2003, pp. 247–256.

[6] S.C. Cheung, J. Kramer, Tractable flow analysis for anomaly detection in distributed programs, in: Proceedings of the 4th European Software Engineering Conference (ESEC 1993), Lecture Notes in Computer Science, vol. 717, Springer, Berlin, Germany, 1993, pp. 283–300.

[7] S.C. Cheung, J. Kramer, An integrated method for effective behaviour analysis of distributed systems, in: Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), IEEE Computer Society, Los Alamitos, CA, 1994, pp. 309–320.

[8] S.C. Cheung, J. Kramer, Tractable dataflow analysis for distributed systems, IEEE Transactions on Software Engineering 20 (8) (1994) 579–593.

[9] S.C. Cheung, J. Kramer, Context constraints for compositional reachability analysis, ACM Transactions on Software Engineering and Methodology 5 (4) (1996) 334–377.

[10] J.M. Cobleigh, L.A. Clarke, L.J. Osterweil, FLAVERS: a finite state verification technique for software systems, IBM Systems Journal 41 (1) (2002) 140–165.

[11] M. Diaz, Petri Nets: Fundamental Models, Verification and Applications, Wiley, Hoboken, NJ, 2009.

[12] E. Duesterwald, R. Gupta, M.L. Soffa, A practical framework for demand-driven interprocedural data flow analysis, ACM Transactions on Programming Languages and Systems 19 (6) (1997) 992–1030.

[13] M.B. Dwyer, L.A. Clarke, J.M. Cobleigh, G. Naumovich, Flow analysis for verifying properties of concurrent software systems, ACM Transactions on Software Engineering and Methodology 13 (4) (2004) 359–430.

[14] R. Enders, T. Filkorn, D. Taubner, Generating BDDs for symbolic model checking in CCS, Distributed Computing 6 (3) (1993) 155–164.

[15] N. De Francesco, A. Santone, G. Vaglini, A user-friendly interface to specify temporal properties of concurrent systems, Information Sciences 177 (1) (2007) 299–311.

[16] M.G. Gouda, C.-K. Chang, Proving liveness for networks of communicating finite state machines, ACM Transactions on Programming Languages and Systems 8 (1) (1986) 154–180.

[17] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, Reading, MA, 2003.

[18] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, IEEE Computer Society, Los Alamitos, CA, 1990.

[19] S. Iyer, S. Ramesh, Apportioning: a technique for efficient reachability analysis of concurrent object-oriented programs, IEEE Transactions on Software Engineering 27 (11) (2001) 1037–1056.

[20] E.B. Johnsen, O. Owea, I.C. Yua, Creol: a type-safe object-oriented model for distributed concurrent systems, Theoretical Computer Science 365 (1–2) (2006) 23–66.

[21] P.V. Koppol, R.H. Carver, K.-C. Tai, Incremental integration testing of concurrent programs, IEEE Transactions on Software Engineering 28 (6) (2002) 607–623.

[22] S. Lu, S. Park, E. Seo, Y. Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII), ACM, New York, NY, 2008, pp. 329–339.

[23] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Norwell, MA, 1993.

[24] R. Meolic, T. Kapus, Z. Brezočnik, ACTLW: an action-based computation tree logic with unless operator, Information Sciences 178 (6) (2008) 1542–1557.

[25] G. Naumovich, L.A. Clarke, Classifying properties: an alternative to the safety-liveness classification, in: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2000/FSE-8), ACM, New York, NY, 2000, pp. 159–168.

[26] W. Peng, S. Puroshothaman, Data flow analysis of communicating finite state machines, ACM Transactions on Programming Languages and Systems 13 (3) (1991) 399–442.

[27] J.H. Reif, S.A. Smolka, The complexity of reachability in distributed communicating processes, Acta Informatica (3) (1988) 333–354.

[28] J.H. Reif, S.A. Smolka, Data flow analysis of distributed communicating processes, International Journal of Parallel Programming 19 (1) (1990) 1–30.

[29] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1998.

[30] A.W. Roscoe, Understanding Concurrent Systems, Springer, London, UK, 2010.

[31] A. Sampaio, S. Nogueira, A. Mota, Compositional verification of input-output conformance via CSP refinement checking, in: Formal Methods and Software Engineering, Lecture Notes in Computer Science, vol. 5885, Springer, Berlin, Germany, 2009, pp. 20–48.

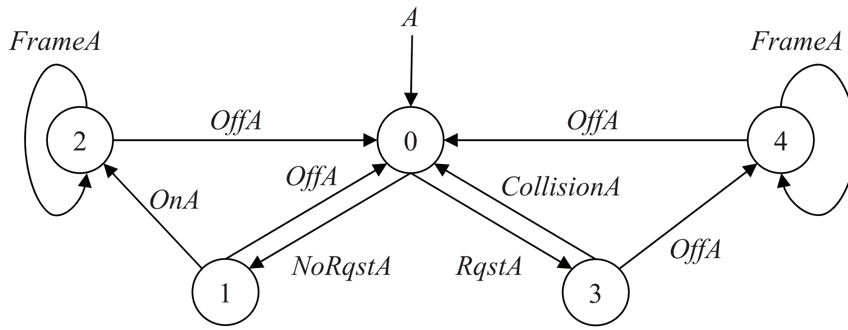
[32] R. Sharp, Principles of Protocol Design, Springer, Berlin, Germany, 2008.

[33] M.M. Strout, B. Kreaseck, P.D. Hovland, Data-flow analysis for MPI programs, in: Proceedings of International Conference on Parallel Processing (ICPP 2006), IEEE Computer Society, Los Alamitos, CA, 2006, pp. 175–184.

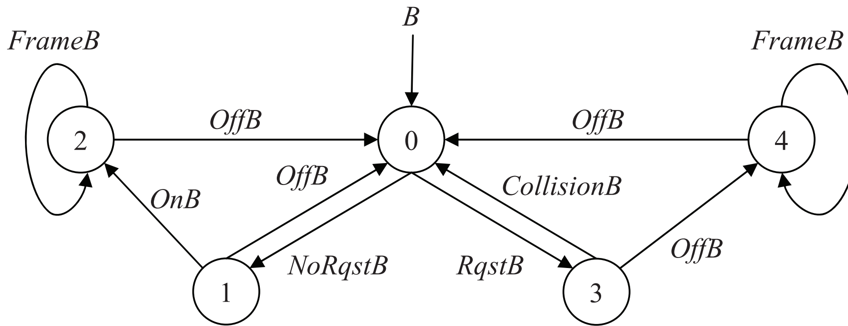
[34] R.N. Taylor, A general-purpose algorithm for analyzing concurrent programs, Communications of the ACM 26 (5) (1983) 361–376.

[35] J.J.P. Tsai, E.Y.T. Juan, Model and heuristic technique for efficient verification of component-based software systems, in: Proceedings of the 1st IEEE International Conference on Cognitive Informatics (ICCI 2002), IEEE Computer Society, Los Alamitos, CA, 2002, pp. 59–68.

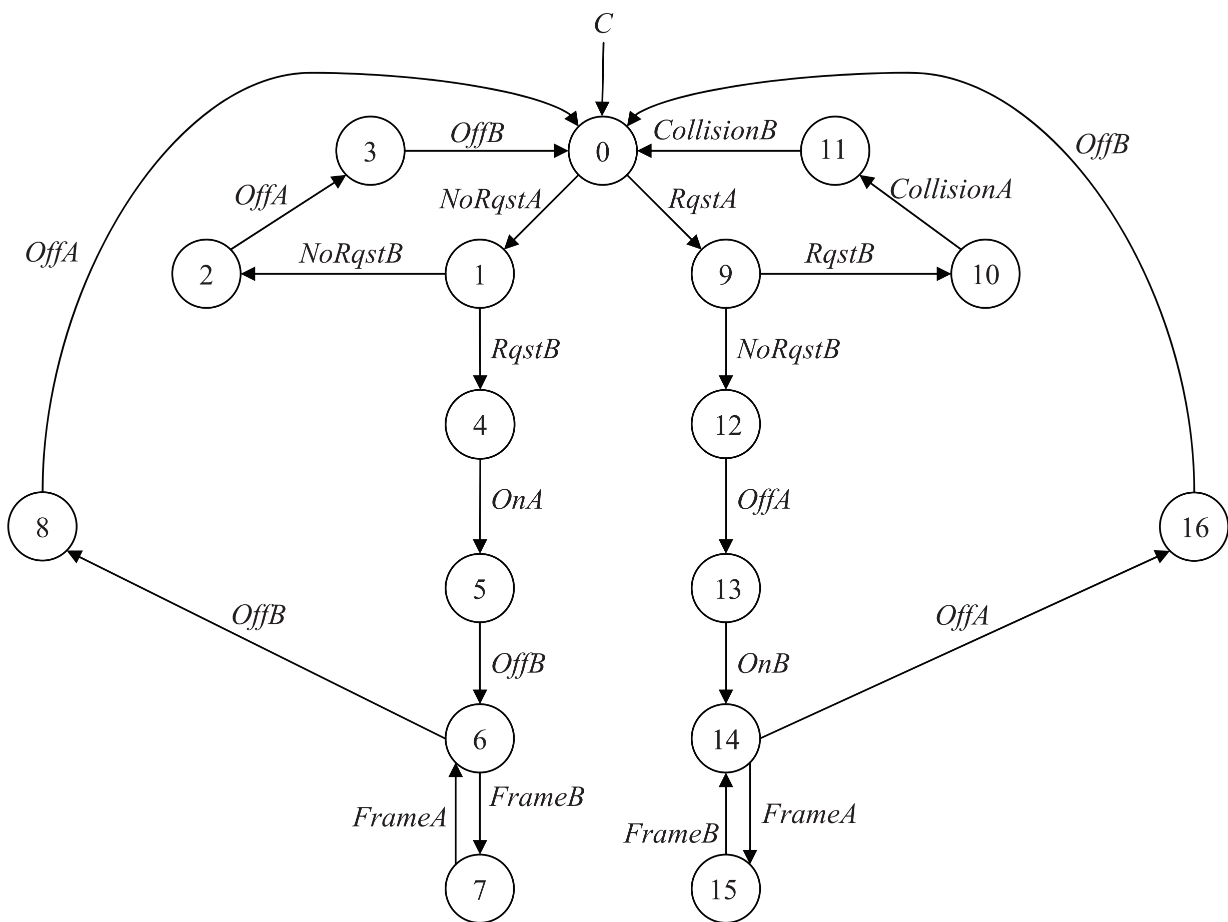
[36] Y. Yamauchi, D. Bein, T. Masuzawa, L. Morales, I.H. Sudborough, Calibrating embedded protocols on asynchronous systems, Information Sciences 180 (10) (2010) 1793–1801.



(a) FSM for Machine *A*



(b) FSM for Machine *B*



(c) FSM for Controller

Figure 12: FSMs of CSMA/CD protocol.

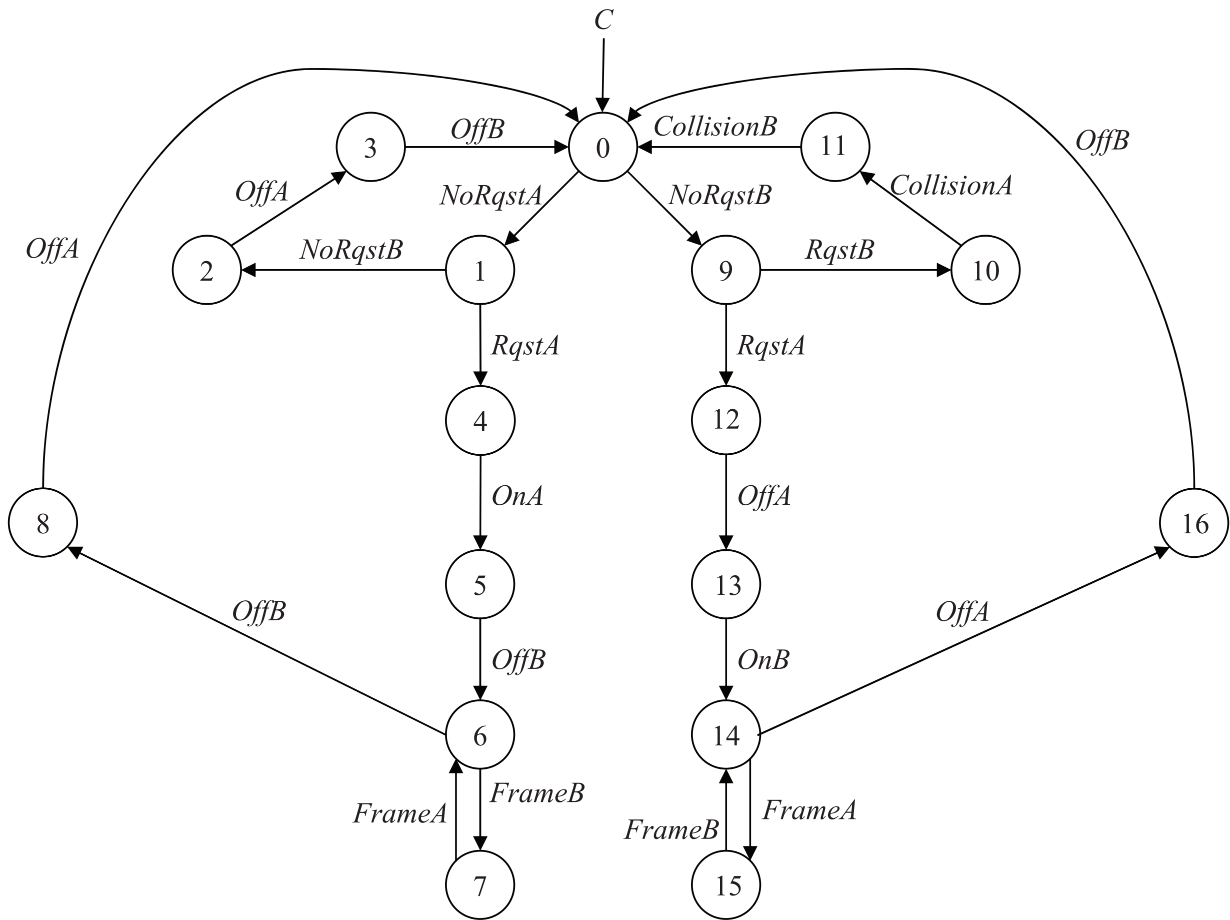


Figure 13: FSM of faulty controller in CSMA/CD protocol.