Learning to Verify Safety Properties

Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, Gul Agha
Department of Computer Science,
University of Illinois at Urbana-Champaign, Urbana, IL, USA.
{vardhan,ksen,vmahesh,agha}@cs.uiuc.edu

Abstract. We present a novel approach for verifying safety properties of finite state machines communicating over unbounded FIFO channels that is based on applying machine learning techniques. We assume that we are given a model of the system and learn the set of reachable states from a sample set of executions of the system, instead of attempting to iteratively compute the reachable states. The learnt set of reachable states is then used to either prove that the system is safe or to produce a valid execution of the system leading to an unsafe state (*i.e.* a counterexample). We have implemented this method for verifying FIFO automata in a tool called LEVER that uses a regular language learning algorithm called RPNI. We apply our tool to a few case studies and report our experience with this method. We also demonstrate how this method can be generalized and applied to the verification of other infinite state systems.

1 Introduction

Software systems are often abstracted as infinite state systems at the design and modeling stage. A popular model for a variety of such systems comprises of finite state machines communicating over unbounded FIFO (first in first out) channels (*FIFO automata*). Examples of such abstraction include: networking protocols where unbounded buffers are assumed, languages like Estelle and SDL (Specification and Description Language) in which processes have infinite queue size, distributed systems and various *actor* systems. A generic task in the automated verification of safety properties of any system is to compute a representation for the set of reachable states. For finite state systems, this is typically accomplished by doing an exhaustive exploration of the state-space. However, for infinite state systems, except in a few special classes [19], exhaustive exploration of the state space is impossible; and in fact the verification problem in general can shown to be undecidable.

We develop a novel *machine learning* based procedure for verifying safety properties of FIFO automata. We assume that the reachable states of the system is a regular set (or is contained in a regular set) which is a fixpoint with respect to the transition relation of the system. Instead of trying to compute the set of reachable states iteratively, we *learn* the set of reachable states from sample runs of the system being verified. If the set of reachable states turns out to be closed (fixpoint) under the transition relation of the system and does not contain any unsafe state, we deem the system to be correct. On the other hand, unsafe states in the learned reachable set are used to obtain executions that might lead to the unsafe state (*i.e.* a counterexample). The counterexample may or may not represent valid executions of the system because we may over-generalize while learning. If the counterexample

turns out to be valid then we have discovered a bug in our system. On the other hand, if the counterexample is invalid, then we use it to refine the learnt set of reachable states. We repeat the process until we have either proved the system to be correct or discovered a buggy execution. Figure 1 shows the overall framework of the *learning to verify* procedure.

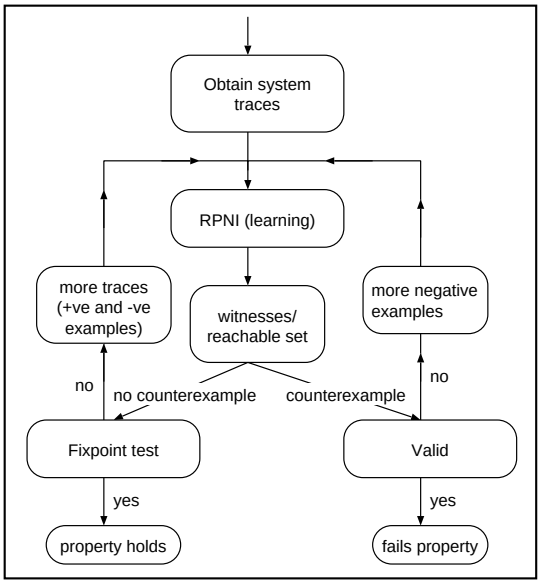

Fig. 1. Learning to verify procedure

Similar to regular model checking [10], we represent states of the system by strings and expect the set of reachable states of practical systems to be structured and representable as regular languages. We then use a modified version of the algorithm RPNI [26, 15], that learns regular languages, to identify the set of reachable states. We show that the algorithm that we present is a *complete verification* method for systems with regular reachable sets[1]; in other words, for such systems we will eventually either find a buggy execution that violates the safety property, or will successfully prove that no unsafe state is reachable. We have implemented our algorithm in Java, and demonstrated the feasibility of this method by running the implementation on simple examples and network protocols such as the alternating-bit protocol and the sliding window protocol. Our approach is complementary to previous methods for algorithmic verification that have been proposed, and we present examples of FIFO automata that our method successfully verifies but on which other approaches fail (see Section 7 on related work). We also

[1] Actually, we require the reachable states along with witness executions to form a regular language; for a precise condition see Section 4.

give the requirements under which classes of infinite state systems other than FIFO automata can be verified using the learning approach.

The rest of the paper is organized as follows. We first (Section 2) introduce the learning framework and recall the definition of FIFO automata. In Section 3, we describe the verification procedure for FIFO automata. Then, (in Section 4) we give the proof of soundness of the method and completeness under certain conditions. In Section 5, we generalize from FIFO automata to give a learning based paradigm for verification of infinite state systems. Finally, in Section 6, we give details of our Java implementation with experimental results, and conclude with overall lessons learnt and directions for future research (Section 8). Relationship of our work to previous research on the algorithmic verification of infinite state systems in general, and FIFO automata in particular, is deferred to Section 7.

2 Preliminaries

In this section, we describe machine learning framework that we use and recall the definition of FIFO automata.

2.1 Learning framework

A learning algorithm is usually set in a framework which describes the types of input data and queries available to the learner. The framework sometimes includes a knowledgeable teacher (student-teacher framework [3]) which provides answers to membership queries (whether a given example belongs to a given concept) and equivalence queries (whether a given hypothesis matches the concept). However, in practice, such a teacher may not be available. Therefore, a more general framework for learning assumes that the learner is simply given examples included in the target concept (positive examples) and examples not included in the target concept (negative examples). One such framework called *language identification in the limit* was introduced by Gold [20] for inference of infinitary languages. In Gold's framework, the learner is given successively larger sequences of positive and negative examples. If the learner is able to converge on the target language after being given a sufficiently large sample of positive and negative examples, it is said to identify the language in the limit. The sample that is needed to guarantee this identification is said to be *characteristic*.

In our setting, we do not have access to a teacher which can answer equivalence queries. Therefore, we restrict ourselves to Gold's framework of identification in the limit. Moreover, in this paper we focus on learning of regular languages; based on the experience of *regular model checking* [10], regular languages are often sufficient to capture the behavior of an interesting class of infinite state systems. A well-known algorithm for the inference of regular languages in Gold's framework is RPNI (regular positive and negative inference) [26, 15]. In this algorithm, the target concept to be learned is a *deterministic finite automata* (DFA) which accepts a regular language. The input consists of a set of positive samples S^+ accepted by the target DFA and a set of negative samples S^- rejected by the target DFA. We use a modified version of the RPNI algorithm which is described in more detail later.

2.2 FIFO Automata

A FIFO automaton [18] is a 6-tuple $(Q, q_0, C, M, \Theta, \delta)$ where Q is a finite set of *control states*, $q_0 \in Q$ is the initial control state, C is a finite set of *channel names*, M is a finite alphabet for contents of a channel, Θ is a finite set of transitions names, and $\delta : \Theta \to Q \times ((C \times \{?, !\} \times M) \cup \{\tau\}) \times Q$ is a function that assigns a *control transition* to each transition name. For a transition name θ, if the associated control transition $\delta(\theta)$ is of the form $(q, c?m, q')$ then it denotes a *receive* action, if it is of the form $(q, c!m, q')$ it denotes a *send* action, and if it is of the form (q, τ, q') then it denotes an *internal* action. The channels are considered to be perfect and messages sent by a sender are received in the order in which they were sent. The formal operational semantics, given by a labelled transition systems, is defined below.

A FIFO automaton $F = (Q, q_0, C, M, \Theta, \delta)$ defines a labelled transition system $\mathcal{L} = (S, \Theta, \to)$ where

- The set of states $S = Q \times (M^*)^C$; in other words, each state of the labelled transition system consists of a control state q and a C-indexed vector of words w denoting the channel contents.
- If $\delta(\theta) = (q, c?m, q')$ then $(p, w) \xrightarrow{\theta} (p', w')$ iff $p = q$, $p' = q'$ and $w = w'[c \mapsto m \cdot w'[c]]$
- If $\delta(\theta) = (q, c!m, q')$ then $(p, w) \xrightarrow{\theta} (p', w')$ iff $p = q$, $p' = q'$ and $w' = w[c \mapsto m \cdot w[c]]$
- If $\delta(\theta) = (q, \tau, q')$ then $(p, w) \xrightarrow{\theta} (p', w')$ iff $p = q$, $p' = q'$ and $w' = w$.

Here $w[i \mapsto s]$ stand for the C-indexed vector which is identical to w for all channels except i, where it is s; $w[i]$ denotes the contents of the channel i. We say $(p, w) \to (p', w')$ provided there is some θ such that $(p, w) \xrightarrow{\theta} (p', w')$. As usual, $\to^*$ will denote the reflexive transitive closure of $\to$. For $\sigma = \theta_1\theta_2 \cdots \theta_n \in \Theta^*$, we say $(p, w) \xrightarrow{\sigma} (p', w')$ when there exist states $(p_1, w_1) \ldots (p_{n-1}, w_{n-1})$ such that $(p, w) \xrightarrow{\theta_1} (p_1, w_1) \xrightarrow{\theta_2} \cdots (p_{n-1}, w_{n-1} \xrightarrow{\theta_n} (p', w')$. The trace language of the FIFO automaton is

$$L(F) = \{\sigma \in \Theta^* \mid \exists s = (p, w).\ s_0 \xrightarrow{\sigma} s\}$$

where $s_0 = (q_0, (\epsilon, \ldots, \epsilon))$, i.e., the initial control state with no messages in the channels.

3 Verification procedure

We now describe the verification procedure in detail with reference to FIFO automata.

The central idea in our approach is to learn the set of reachable states instead of computing it by iteratively applying the transition relation. Once the set of reachable states is learnt, we can verify if the safety property is violated by checking if an unsafe state is among the set of reachable states. However, in order to ensure the soundness of our results, we need a mechanism to check if the output of the learning algorithm is indeed correct. Observe that if the set of states learnt is closed under the transition relation then it means that the learnt set of states contains

all the reachable states, and if, in addition, none of the states in the learnt set are unsafe then we can conclude that the system satisfies the safety property. On the other hand, if one of the states in the learnt set is unsafe, then we need a mechanism to check whether the learning algorithm over-generalized, i.e., added states that are not reachable. One way to accomplish this is by producing a candidate execution to the unsafe state, and checking if that execution is indeed a valid execution of the system. Therefore, instead of learning the set of reachable states directly, we learn a language which allows us to identify both the reachable states and witnesses to these in terms of transitions executed by the system.

Let us now consider the language which can allow us to find both reachable states and their witnesses. The first choice that comes to mind is the language of the traces, $L(F)$. Since each trace uniquely determines the final state in the trace, $L(F)$ has the information about the states that can be reached. While it is easy to compute the state s such that $s_0 \xrightarrow{\sigma} s$ for a *single* trace σ, it is not clear how to obtain the set of states reached, given a *set of traces*. In fact, even if $L(F)$ is regular, there is no known algorithm to compute the corresponding set of reachable states of the labelled transition system.[2] The main difficulty is that determining if a receive action can be executed depends non-trivially on the sequence of actions executed before the receive. We overcome this difficulty by annotating the traces in a way that makes it possible to compute the set of reachable states.

Trace Annotation for FIFO: Consider a set $\overline{\Theta}$ of *co-names* defined as follows:

$$\overline{\Theta} = \{\overline{\theta} \mid \theta \in \Theta \text{ and } \delta(\theta) \neq \tau\}$$

In other words, for every send or receive action in our FIFO automaton, we introduce a new transition name with a "bar". We say $s \xrightarrow{\overline{\theta}} s'$ if $s \xrightarrow{\theta} s'$; executions over sequences in $(\Theta \cup \overline{\Theta})^*$ are defined naturally. The intuition of putting the annotation of a "bar" on some transitions of a trace is to indicate that the message sent or received as a result of this transition does not play a role in the channel contents of the final state. In other words, a "barred" transition $\overline{\theta}$ in an annotated trace of the system denotes either a message sent that will later be received, or the receipt of a message that was sent earlier in the trace. Thus, annotated traces of the automaton will be obtained by marking send-receive pairs in a trace exhibited by the machine. Let $\mathcal{A}$ be the function that correctly annotates an execution to produce a string over $\Sigma = \Theta \cup \overline{\Theta}$. Observe, that each execution is annotated uniquely, or to put it formally, $\mathcal{A}$ is an injective function. The *annotated trace language* of the automaton F is $AL(F) = \{\mathcal{A}(t) \mid t \in L(F)\}$ and consists of all strings in $(\Theta \cup \overline{\Theta})^*$ that denote correctly annotated traces of F. For example, consider the FIFO automaton shown in Figure 2(a). Some of the words in $AL(F)$ are: θ_1, $\theta_1\theta_1$, $\theta_1\theta_1\theta_1$, $\overline{\theta_1\theta_2}$, $\overline{\theta_1\theta_2}\theta_1$.

Finding reachable states from annotated traces: Since our objective is to identify the reachable region, we need a way to find the reachable states corresponding to a set of annotated traces. For a channel c, consider a function h_c :

[2] This can sometimes be computed for simple loops using meta-transitions.

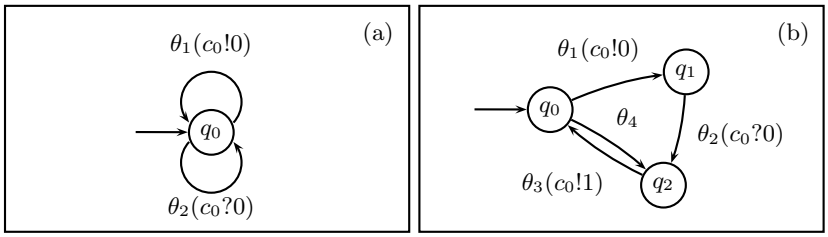

Fig. 2. Example FIFO automata. θ_4 does not change channel contents.

$(\Theta \cup \overline{\Theta}) \to M^*$ defined as follows:

$$h_c(t) = \begin{cases} m \text{ if } t \in \Theta \text{ and } \delta(t) = c!m \\ \epsilon \quad \text{otherwise} \end{cases}$$

Let h_c also denote the unique homomorphism from $(\Theta \cup \overline{\Theta})^*$ to M^* that extends the above function. Given an annotated trace ρ, the contents of channel c in the final state are clearly given by $h_c(\rho)$.

FIFO automata with one channel: Let $F = (Q, q_0, \{c_0\}, M, \Theta, \delta)$ be a single channel FIFO automaton, with c_0 being the only channel. As usual $s_0 = (q_0, \epsilon)$ will denote the starting state of F. Given a set of annotated traces L, let $L_q \subseteq L$ be the set of annotated traces in L whose last transition ends in control state q. Now the set of states reached (by traces in L) is given by

$$\mathcal{R}(L) = \{(q, m) \mid q \in Q \text{ and } m \in h_{c_0}(L_q)\}$$

For a regular set L, then it can be seen that L_q is regular, and $\mathcal{R}(L)$ can be computed by a simple homomorphism, and so $\mathcal{R}(L)$ is regular.

Multi-channel FIFO automata: Consider a FIFO automaton $F = (Q, q_0, C, M, \Theta, \delta)$ communicating over channels $C = \{c_0, c_1, \ldots c_k\}$. Now the set of states reached (by traces in L) is given by

$$\mathcal{R}_m(L) = \{(q, (h_{c_0}(\sigma), h_{c_1}(\sigma), \ldots, h_{c_k}(\sigma))) \mid \\ q \in Q \text{ and } \sigma \in L_q\}$$

As we will see shortly, we need a test for inclusion for the reachable states corresponding to a set of annotated traces. In this respect, we cannot hope to work with $\mathcal{R}_m(L)$, since as soon as have even two channels, given a regular L, $\mathcal{R}_m(L)$ can be seen to be a rational relation for which inclusion is undecidable [6]. However, if we compute the contents of each channel independently of others, we can compute an upper approximation of $\mathcal{R}_m(L)$ as follows:

$$\mathcal{R}(L) = \bigcup_{q \in Q} \{q\} \times h_{c_0}(L_q) \times h_{c_1}(L_q) \cdots h_{c_k}(L_q)$$

It can be easily seen that $\mathcal{R}(L)$ is a regular language if L is regular. In general, $\mathcal{R}_m(L) \subseteq \mathcal{R}(L)$, however for many FIFO systems encountered in practice (most network protocols like Alternating Bit Protocol, Sliding Window Protocol), this gives the exact reachable region when applied to $AL(F)$, *i.e.* $\mathcal{R}_m(AL(F)) = \mathcal{R}(AL(F))$. We show later that this is sufficient for the applicability of our learning approach.

Recovering a witness from an unsafe state: If the reachable states corresponding to a learned set of annotated traces have a nonempty intersection with the set of "unsafe" states (which violate the safety property), we would like to extract a sequence of transitions of the system which witnesses the reachability of some unsafe state. The motivation is that such a sequence can then be used as a counterexample demonstrating the violation of the safety property or a negative example for the learning algorithm.

We assume that for each control state q, we are given the unsafe channel contents as a product of regular languages $U(q, c_0), U(q, c_1), \ldots, U(q, c_n)$ corresponding to channels $0 \ldots n$, *i.e.*, the unsafe states are given by $S_u = \bigcup_{q \in Q} \{(q, u_0, u_1, \ldots, u_n) \mid u_i \in U(q, c_i)\}$. Given a regular set of annotated traces, L, recall that $L_q \subseteq L$ represents the set of annotated traces in L whose last transition ends in control state q. For each control state q, for each channel i, we can find the intersection of $h_{c_i}(L_q)$ and $U(q, c_i)$ and calculate the traces $L_{q_i} \in L_q$ such that $h_{c_i}(L_{q_i}) = h_{c_i}(L_q) \cap U(q, c_i)$. Intuitively, this gives us annotated traces which lead to a potential unsafe configuration for channel c_i. Now, if the intersection $\bigcap_{i \in 0 \ldots n} L_{q_i}$ is non empty, an annotated trace t in this intersection leads to an unsafe configuration for each channel and hence an unsafe state in S_u. Let us call $\mathcal{W}$ the function which outputs t given L and S_u.

From annotated trace to system execution: In order to convert $\mathcal{W}(L) \in \Sigma^*$ into a sequence of transitions, we need a way to extract the presumed system execution from a given annotated trace. Essentially, we want a substitution $RevA : \Sigma \mapsto \Theta^*$ which "reverses" the annotation $\mathcal{A}$. This can be done simply by removing the "bars" on the annotated trace. Formally, we can define $RevA(\overline{\theta}) = RevA(\theta) = \theta$ for all $\theta \in \Theta$. Extending $RevA$ to strings in the usual way, it can be seen that that $RevA(\mathcal{A}(t)) = t$.

Verification algorithm We are now ready to formally describe the *learning to verify* procedure which was introduced in Figure 3. We first collect positive and negative examples of labels in Σ^* as follows. A set T of sequences of transitions that can be exhibited by the system is obtained by invoking a function $GetTraces$. Positive examples, S^+ are simply the correct "annotations" which put bars on the send-receive pairs in the strings in T, *i.e.* $S^+ = \{\mathcal{A}(t) \mid t \in T\}$. There are three sets of negative examples. The first set $S_1^- = \{t\theta_d \mid t \in T \text{ and } \theta_d \text{ is a disabled transition}\}$ consists of sequences of transitions extended by a disabled transition (a transition that cannot be taken at a certain state). The second set $S_{2a}^- = \{\sigma \in \Sigma^* \mid \exists t \in T \text{ such that } RevA(\sigma) = t \text{ and } \sigma \neq \mathcal{A}(t)\}$ corresponds to "incorrect" annotations. Notice that since $\mathcal{A}$ is injective, all annotations of a trace $t \in T$ other than $\mathcal{A}(t)$

algorithm *learnToVerify*
Input:
F : model of system,
S_u : regular set of "unsafe states"
Output: Property valid OR
path to an unsafe state
begin
$S^-_{2b} = \emptyset$
$(S^+, S^-_1, S^-_{2a}) =$*GetTraces*()
while(true)do
$L =$ *modifiedRPNI*$(S^+, S^-_1, S^-_{2a} \cup S^-_{2b})$
if $\mathcal{R}(L) \cap S_u \neq \emptyset$
$l_c = \mathcal{W}(L, S_u)$
if $RevA(l_c)$ valid execution of F
Output $RevA(l_c)$; stop
else
$S^-_{2b} = S^-_{2b} \cup l_c$
else
if $\mathcal{R}(L)$ is a fixpoint
Output "Property holds"; stop
else
$T_{\text{new}} =$ *GetTraces*()
add T_{new} to (S^+, S^-_1, S^-_{2a})
end

algorithm *modifiedRPNI*
Input: $S^+ \in \Sigma^*$, $S^-_1 \in \Theta^*$, $S^-_2 \in \Sigma^*$
Output: a regular language L
begin
$D \leftarrow PTA(S^+)$
for $i = 2$ to $|D|$ do
for $j = 1$ to $i - 1$ do
if q_i, q_j not merged with smaller state then
$D' \leftarrow$ *merge*(D, q_i, q_j)
$D' \leftarrow$ *determinize*(D', q_j)
$D'' \leftarrow RevA(D')$; all states in D'' made final
if *compatible*(D'', S^-_1) && *compatible*(D', S^-_2)
D = D';exit j-loop
return language defined by D
end

algorithm *determinize*
Input: A, x; **Output:** A
begin
for any $x \xrightarrow{\theta} x_1, x \xrightarrow{\theta} x_2$ and $x_1 \neq x_2$
$A \leftarrow$*merge*(A, x_1, x_2)
$A \leftarrow$ *determinize*(A, smaller of x_1, x_2)
return A
end

Fig. 3. Learning to verify algorithm

cannot be exhibited by the system. The third set, S^-_{2b}, is a collection of spurious counterexamples; initially this is empty.

The positive and negative examples are given to a learning algorithm based on RPNI. Similar to RPNI, this algorithm first constructs a *prefix tree automata* (PTA) from S^+ which is simply a collection of the strings as paths with common prefixes merged together. Each state in the PTA is associated with the string generated by following the path to that state from the initial state. The states are assigned numbers according to the standard ordering[3] imposed by the associated strings. The learning algorithm attempts to generalize from the positive examples by merging states in the PTA in a *specific order*: for i going from 1 to the largest state in the PTA, it attempts to merge q_i with all states less than q_i in ascending order. A merge may cause non-determinism which is removed by further merges using the operation *determinize* which results in a finite automaton D'. Another finite automaton D'' is obtained from D' by applying the substitution $RevA$ and making all states final. If D'' is compatible with the negative set S^-_1 (all strings in S^-_1 are rejected by D'') and D' is compatible with the negative set $S^-_2 = S^-_{2a} \cup S^-_{2b}$, the merge is accepted. The learning algorithm is essentially the same as the traditional RPNI algorithm except for the the use of the additional kind of negative examples corresponding to S^-_1. For a detailed explanation of the RPNI algorithm itself, the reader is referred to [26, 15].

[3] For $\Sigma = \{a, b\}$, the ordering is $\epsilon, a, b, aa, ab, ba, bb, bb, aaa, \ldots$

Let the output of the modified RPNI algorithm be the regular language L. If $\mathcal{R}(L)$ intersects with the unsafe states S_u, then a counterexample l_c $(= \mathcal{W}(AL(F), S_u))$ is obtained. By attempting to simulate the counterexample on the system, we can check if $RevA(l)$ is executable. If yes, then we have found a real counterexample and are done, otherwise l_c is added to S^-_{2b}. If $\mathcal{R}(L)$ does not intersect with the unsafe states S_u, then it is tested for being a fixpoint under the reachability relation by checking the following condition:

$$\{s_0\} \cup \{s \mid \exists s' \in \mathcal{R}(L).\ s' \rightarrow s\} = \mathcal{R}(L)$$

If it is a fixpoint, we declare that the safety property holds. Otherwise, we get more traces by invoking the function $GetTraces$ (successive calls to this function generate new traces) and continue the learning procedure.

4 Correctness of the verification procedure

The soundness of the procedure is straightforward. For a learned set of traces L, if $\mathcal{R}(L)$ has an empty intersection with the set of unsafe states, S_u, and is a fixpoint under the transition relation, the safety property holds. Any counterexample is finite and gives a supposed execution of the system leading to an unsafe state which can then be automatically checked for validity by simulation of the system.

We can also show completeness (*i.e.* the procedure terminates with the correct answer) under the condition that $AL(F)$ is regular. Then, given a "fair" method of generating the system traces, in the limit, the learning paradigm will either prove that the system satisfies the property or find a valid counterexample. By a fair method, we mean one which will eventually generate any given finite trace. There can be many different ways of generating fair traces, one of the simplest being a breadth first traversal of all traces.

Lemma 1. *If $AL(F)$ is regular, then using any fair strategy for generating traces, in the limit, given a sufficiently large sample, the learning procedure outputs a DFA which generates $AL(F)$.*

Theorem 1. *If $AL(F)$ is regular and $\mathcal{R}(AL(F))$ is the set of all reachable states, then the learning to verify procedure will eventually either prove that the system satisfies the property or find a valid counterexample.*

The proofs of the lemma and the theorem are given in Appendix A. The running time of the algorithm is dependent on the strategy for getting the traces. For a simple breadth-first strategy, in the worst case, the algorithm might need to explore all traces up to a depth D. Here, D is the length of the longest path starting from the initial state in the minimal DFA representing $AL(F)$ (assuming $AL(F)$ is regular). Thus, the running time can be exponential in the size of the DFA for $AL(F)$. However, as discussed in Section 6, we can use some heuristics to prune down the number of traces needed. In practice, for a number of FIFO systems, the learning procedure is able to converge to the correct answer in a fairly small time period which is comparable to other tools.

Note that the conditions required by Theorem 1 are merely sufficient for termination of the learning procedure and the verification procedure can be successfully used for many systems even if $AL(F)$ is not regular. In fact, an important observation is that for a number of systems with nonregular $AL(F)$, there exists a regular subset $L' \subseteq AL(F)$ such that the traces in L' "cover" all the reachable states, *i.e.* $\mathcal{R}(L') = \mathcal{R}(AL(F))$. In other words, every reachable state in F is witnessed by some trace in L'. For example, the set of annotated traces corresponding to the automaton in Figure 2(a) is not regular but the regular language $L' = \theta_1^*$ covers all the reachable states. Note that $\mathcal{R}(L')$ is not an approximation; we are simply content with finding any regular set of annotated traces that can cover the reachable states. In Section 6, we analyze FIFO systems which have a regular $AL(F)$ as well as systems for which $AL(F)$ is not regular but a "covering" $L' \subseteq AL(F)$ is regular. In all cases, the algorithm terminates with the correct reachable set.

5 Generalization to other infinite state systems

The verification procedure described for FIFO automata can be easily generalized to other infinite state systems. The challenge is to identify the alphabet Σ which provides the "annotation" and the functions $\mathcal{A}$, $RevA$, $\mathcal{R}$ and $\mathcal{W}$ which are used by the verification procedure. Notice that the procedure does not assume anything else about FIFO automata other than the above functions. The key properties of Σ needed to make the procedure work are summarized below.

- There exists an *injective* function $\mathcal{A} : L(F) \mapsto \Sigma^*$ which maps a system execution to a sequence of labels in Σ. Recall that $L(F)$ is the language of traces that can be executed by the system F. Let $AL(F) \in \Sigma^*$ be the language $\{\mathcal{A}(t) \mid t \in L(F)\}$.
- There exists a substitution $RevA : \Sigma \mapsto \Theta^*$ which "reverses" the operation $\mathcal{A}$. Extending $RevA$ to strings in the usual way, it must be true that $RevA(\mathcal{A}(t)) = t$. In FIFOs, $RevA$ simply removes the "bars".
- There exists a (computable) function $\mathcal{R}$ such that for a set $L \in \Sigma^*$, $\mathcal{R}(L)$ gives a set of states (supposedly) reached during the execution of the traces in L. It is required that $\mathcal{R}(AL(F))$ must be the exact reachable region of F.
- There exists a (computable) function $\mathcal{W}$ such that for $L \in \Sigma^*$, and a set of "unsafe" states S_u, if $\mathcal{R}(L) \cap S_u \neq \emptyset$ then $\mathcal{W}(L, S_u)$ gives a finite trace $l_c \in L$ which witnesses the reachability of some state in S_u.

It can be easily seen that the proof of correctness of the learning algorithm in Section 4 generalizes to other systems if Σ satisfies the above properties. Thus, we can think of this approach as a "paradigm" for the verification of safety properties of infinite systems.

6 Implementation

We have implemented the verification framework for FIFO automata as part of the LEVER (LEarning to VERify) tool suite available from [23]. The tool is written in Java and implements the *learning to verify* procedure shown in Figure 3. For general

automata related decision procedures, we use the Java package `dk.brics.automata` available from [24]. Currently, the incremental learning approach is not implemented in LEVER, so if an answer to the verification problem is not solved in a particular run, we restart the procedure with more positive samples.

For generating the annotated traces that are used for the positive and negative examples, we use the following strategy. Starting from the initial state, we explore the system states (cross product of the control state and channel contents) in a breadth-first manner. To limit the number of traces generated, we do not distinguish between FIFO states if they have the same control state and same channel contents up to a position d from the start of the channel. We start with $d = 1$ and keep increasing d if more traces are needed. We have seen that this heuristic works quite well in practice to generate sufficient traces for the learning procedure.

We have used LEVER to analyze some canonical FIFO automata verification problems described below.

[Producer Consumer] A simple producer consumer problem with one FIFO channel. The producer can either be in an "idle" or in a "send" state in which it transmits either 0 or 1 to the FIFO channel.

[Data with parity] A simple data communication protocol in which the sender sends data and a parity bit for the number of 1's sent. The receiver uses the parity bit as a simple check for data integrity.

[Resource arbitrator] In this example, two senders wish to broadcast over a shared channel and use a resource manager to arbitrate which one is allowed to use it at any time.

[Alternating bit protocol (ABP)] This consists of a sender and receiver communicating over a data and an acknowledgment channel. We consider a non-lossy version of ABP.

[Sliding window protocol] This is similar to ABP except that the sender can keep multiple data messages in flight. We use a window size of 2 and maximum sequence number also of 2.

"Producer Consumer", "Alternating bit protocol" and "Sliding window protocol" are fairly well-known in the FIFO research community, see for example [28]. For the other two systems, a detailed description is available in Appendix B.

Table 1 shows the results obtained. Here "Samples" is the number of positive samples generated, T is the running time on a 1594 MHz notebook computer with 512 MB of RAM using Java virtual machine version 1.4.1 from Sun Microsystems. In all cases, LEVER terminates with the correct reachable region. We also report the time taken (T_{rmc}) by the regular model checking tool [25] on the same examples. Although a complete comparative analysis with all available tools remains to be done, it can be seen the running time of LEVER is comparable to the regular model checking tool and in fact better for all examples except "Sliding window protocol".[4]

LEVER is still in the prototype stage and we see the current version as a proof of concept of the *learning to verify* approach. We plan to introduce various optimizations and features which should enable LEVER to handle larger and more complex examples.

[4] The encoding of sliding window protocol in the regular model checking tool is slightly different; instead of limiting the window size, the size of the queue is bounded.

	Samples	T	T_{rmc}
Producer Consumer	42	0.4s	3.3s
Data with parity	42	0.5s	12.7s
Resource arbitrator	146	0.7s	33.2
Alternating Bit	1122	4.1s	24.7
Sliding Window	2535	81.2s	78.4

Table 1. Samples and running time

7 Related Work

Verification of infinite state systems: For automatic verification of infinite state systems, the state space has to be represented by symbolic means. Some common representations are: regular sets [10, 1], Presburger formulas [11], Queue Decision Diagrams and Number Decision Diagrams [7], semi-linear regular expressions [18], constrained QDDs [9] and constrained facts [14]. Since an iterative approach of computing the fixpoint for reachability does not terminate for most cases, various mechanisms are used for finding the reachable set. We now discuss some of these techniques and show their relation to our learning approach.

In the approach using *meta-transitions* and *acceleration* [7, 9, 18], a sequence of transitions, referred to as a *meta-transition*, is selected and the effect of its infinite iteration calculated. There is no possibility of a spurious counterexample but it may not be possible to find all meta-transitions needed to compute the exact reachable set. This is complementary to our learning approach, since meta-transitions can be also be incorporated into our learning algorithm.

Another popular approach for FIFO, parametric, integer and stack systems is *regular model checking* [10, 1]. A regular set is used to represent the states and a transducer is used to represent the transition relation. The problem is reduced to finding a finite transducer representing the the infinite composition of this relation. This can be done by subset construction [10]; by using an equivalence relation to merge states [1]; or by "iterating transducers" [13, 8]. However, there are some examples in which even if such a finite transducer exists, the procedure may not be able to converge to it. One such example of a FIFO automaton is shown in Figure 2(b) in Section 3. We used the regular model checking tool from [25] to analyze this example, but both the subset-construction method and method based on equivalence relation failed to terminate even after two hours. A careful analysis revealed that for this example, at each step of the regular model checking method, the intermediate transducer keeps accumulating new states in a pattern that causes its size to grow without bound. On the other hand, our learning-based tool is able to automatically find the reachable set in about fifty milliseconds. We have shown that if the reachable states and the traces witnessing the reachable states form a regular set, our learning method will eventually be able to find it. It is certainly possible that in other examples, transducer construction may be able to find the reachable region faster. Thus, our approach can be seen as complementary and seen to extend the range of systems that can be automatically analyzed.

An approach for computing the reachable region that is closely related to ours is *widening*. In this approach, the transition relation is applied to the initial configuration some number of times and then by comparing the sets thus obtained,

the limit of the iteration is guessed. A simple widening principle in the context of regular model checking is given in [10] which is extended in [29] for parametric systems. Bultan [11] uses a widening technique for Presburger formulas to enable faster convergence for fixpoint. Bartzis *et al.* [5] present a widening technique for arithmetic automata. At a very high level, both *widening* and our approach use similar ideas. In both methods, based on certain sample points obtained using the transitions, a guess is made for the fixpoint being searched for. Exploring the connections between the two approaches is an interesting direction for future work. One important difference is that widening (except for certain special contexts where it can be shown to be exact) is a mechanism to prove the correctness of a system and cannot be used to prove a system to be incorrect. On the other hand, the approach presented here allows one to both prove a system to be correct and to detect bugs.

Another technique for verification of infinite systems uses bisimulation minimization algorithms [22]. A crucial requirement for the success of bisimulation technique is that the reachable set of equivalence classes be finite. It is easy to construct FIFO automata which fail to satisfy this property, although the automata are quite simple and may be analyzed using our machine learning framework automatically. One such automaton is shown in Figure 2(a).

Use of machine learning for verification: The application of techniques from *machine learning* for verification is relatively new. Peled *et al.* [27] give a method called "Black Box Checking" which is extended by Groce *et al.* [21] as *Adaptive Model Checking*. Briefly, in this method, one starts with a possibly inaccurate model and incrementally updates it using Angluin's [3] query based learning of regular sets. Cobleigh *et al.* [12] also use a variant of Angluin's algorithm to learn the assumptions about the environment to aid compositional verification. Boigelot *et al.* [4] present a technique for constructing a finite state machine that simulates all observable operations of a given reactive program. Ammons *et al.* [2] use machine learning to discover formal specifications of the protocols that a client of an application program interface must observe. Edelkamp *et al.* [16] consider the problem of finding "bad" states in a model as a directed search problem and use AI heuristic search methods to attempt to find these states. Ernst *et al.* [17] have developed a system called *Daikon* which attempts to discover likely invariants in a program by analyzing the values taken by its variables while the program is exercised in a test suite.

Our approach in using the machine learning techniques for verification is unique in that we are not trying to learn an unknown system model but rather the behavior of a system which is already fully described. This is closest in spirit to Ernst *et al.* [17], although the domain of application and objective are completely different. Another difference with other learning methods is that we do not use Angluin's algorithm. Angluin's algorithm needs a teacher which can answer equivalence queries; answering such queries is typically hard to achieve. Instead, we use an algorithm called RPNI [26, 15] which simply needs positive and negative samples of the concept to be learned.

8 Conclusion and future work

We have presented a novel approach based on machine learning to verify finite state machines communicating over unbounded FIFO channels. A prototype implementation, called LEVER, has been developed and the approach has been shown to be feasible in analyzing practical networks protocols like alternating bit and sliding window. We have also shown how this approach can be generalized to analyze infinite state systems other than FIFO automata.

There are many promising directions for future work. The procedure can be applied to other systems such as: automata with unbounded integers; real-time and hybrid systems; parameterized systems; counter automata; and push-down automata with multiple stacks. The approach could be extended to handle not only safety properties but also liveness and more general temporal properties. The feasibility of using this approach for probabilistic systems could also be studied. For the machine learning aspect, algorithms can be developed to learn classes that are more expressive than regular languages. Another interesting direction is to investigate the feasibility of using machine learning even for finite state systems. Other avenues for future work include research for getting better execution traces and practical optimizations and engineering for the LEVER tool.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *Computer-Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 236–248. Springer, 2003.
2. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16, Jan. 2002.
3. D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, Nov. 1987.
4. B. Boigelot and P. Godefroid. Automatic synthesis of specifications from the dynamic observation of reactive programs. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 321–334, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.
5. C. Bartzis and T. Bultan. Widening arithmetic automata. In *Computer Aided Verification'04 (to appear)*, 2004.
6. J. Berstel. *Transductions and Context-Free-Languages*. B.G. Teubner, Stuttgart, 1979.
7. B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Collection des Publications de la Faculté des Sciences Appliquées de l'Université de Liége, 1999.
8. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV: International Conference on Computer Aided Verification*, 2003.
9. A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1–2):211–250, June 1999.
10. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

11. T. Bultan. *Automated symbolic analysis of reactive systems*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, Md., 1998.
12. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 331–346, 2003.
13. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. *Journal of Logic and Algebraic Programming*, 52-53:109–127, 2002.
14. G. Delzanno and A. Podelski. Model checking in CLP. *LNCS*, 1579:223–239, 1999.
15. P. Dupont. Incremental regular inference. In *Proceedings of the 3rd International Colloquium on Grammatical Inference (ICGI-96): Learning Syntax from Sentences*, volume 1147 of *LNAI*, pages 222–237, Berlin, September 1996. Springer.
16. S. Edelkamp, A. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-based Validation of Intelligence*, 2001.
17. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering (ICSE'99)*, pages 213–224, 1999.
18. A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1–31, 2003.
19. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1–2):63–92, 2001.
20. E. M. Gold. Language indentification in the limit. *Inform. Control*, 10:447–474, 1967.
21. A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 357–371, 2002.
22. D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In ACM, editor, *Proceedings of the 24th annual ACM Symposium on Theory of Computing*, pages 264–274, New York, NY, USA, 1992. ACM Press.
23. LEVER. Learning to verify tool. `http://osl.cs.uiuc.edu/~vardhan/lever.html`, 2004.
24. A. Møller. `dk.brics.automaton`. `http://www.brics.dk/~amoeller/automaton/`, 2004.
25. M. Nilsson. `Regular model checking tool`. `http://www.regulalrmodelchecking.com`, 2004.
26. J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.
27. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE/PSTV, Beijing, China*, 1999.
28. A. S. Tanenbaum. *Computer Networks, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ, 1989.
29. T. Touili. Regular model checking using widening techniques. In *ENTCS*, volume 50. Elsevier, 2001.

A Proof of correctness of the learning procedure

For a regular language $L \in \Sigma^*$, the prefixes are defined as $Pr(L) = \{u \mid \exists v, uv \in L\}$. Let $L/u = \{v \mid uv \in L\}$ denote the right-quotient of L by u. The standard order of strings in Σ^* is denoted by $<$, *e.g.* for $\Sigma = \{a, b\}$, the ordering is $\epsilon, a, b, aa, ab, ba, bb, bb, aaa, \ldots$

Definition 1. *The set of* short prefixes *of a language L is defined as follows:*

$$Sp(L) = \{x \in Pr(L) \mid \nexists u \in \Sigma^* \text{ with } L/u = L/x \text{ and } u < x\}$$

Given a canonical (minimal) DFA F for a regular language L, every state of F corresponds to an unique element in the set $Sp(L)$ and F as many states as elements in $|Sp(L)|$.

Definition 2. *The kernel $N(L)$ is defined as follows:*

$$N(L) = \{\epsilon\} \cup \{xa \mid x \in Sp(L), a \in \Sigma, xa \in Pr(L)\}$$

The kernel is an extension of a short prefix by a transition in the canonical DFA. Note that $Sp(L) \subseteq N(L)$. Thus, the set of short prefixes represent the states of a canonical DFA and kernel represents the set of transitions of a canonical DFA.

Lemma 2. *If $AL(F)$ is regular, then using any fair strategy for generating traces, in the limit, given a sufficiently large sample (S^+, S_1^-, S_2^-), the modified RPNI algorithm outputs a DFA which generates $AL(F)$.*

Proof (Sketch). Let D be the canonical DFA which generates $AL(F)$. Let Sp and N respectively denote the set of "short prefixes" and "kernel" of $AL(F)$. Since the short prefixes and kernel are finite, given a fair strategy of generating the traces, we are guaranteed to eventually get a set of positive examples, S^+, whose prefixes include all short prefixes and kernel strings. It is known from the proof of RPNI [15], that if we have such positive examples, then the ordered merge used by RPNI outputs a DFA isomorphic to D provided that $\forall x \in Sp, \forall y \in N$ if x and y are *non-equivalent*, *i.e.*, if $AL(F)/x \neq AL(F)/y$, then there are enough negative examples in S_1^- and S_2^- to prevent a merge of x and y. We show that in the limit we are guaranteed to get enough negative examples. Consider some $x \in Sp$ and some $y \in N$ such that $AL(F)/x \neq AL(F)/y$. First consider the case that there is a string u such that $xu \in AL(F)$ but $yu \notin AL(F)$. Recalling that $L(F)$ is the language of all allowed sequences of transitions, there are two possibilities why $yu \notin AL(F)$:

- $RevA(yu) \in L(F)$ but $\mathcal{A}(RevA(yu)) \neq yu$. Intuitively, the sequence of transitions is correct but they have a wrong "annotation". By a fair strategy, we get negative examples S_{2a}^- corresponding to all wrong "annotations" of valid transition sequences, hence in the limit we will have some negative example to detect this situation.
- $RevA(yu) \notin L(F)$. Let v be the minimal prefix of u such that $RevA(xv) \in L(F)$ but $Rev(yv) \notin L(F)$. Since any string shorter than $Rev(yv)$ is in $L(F)$, $Rev(yv)$ is a sequence of valid transitions extended by a disabled transition. By using a

fair strategy, we will eventually get a negative example in S_1^- equal to $Rev(yv)$. When we check for compatibility with S_1^-, we set all states final in the DFA obtained after merge. This enables the detection of the case that $RevA(xv) \in L(F)$ but $Rev(yv) \notin L(F)$

The other possibility that $yu \in AL(F)$ but $xu \notin AL(F)$ is handled in the same manner.

Theorem 2. *If $AL(F)$ is regular and $\mathcal{R}(AL(F))$ is the set of all reachable states, then the learning to verify procedure will eventually either prove that the system satisfies the property or find a valid counterexample.*

Proof (Sketch). Since we use a fair strategy for generating traces, by Lemma 1 if we keep generating more traces, eventually we will learn $AL(F)$. In this case, $\mathcal{R}(AL(F))$ would be a fix point and if the safety property holds, the procedure does not find a counterexample and proves that the system satisfies the property. If the safety property does not hold then the reachable states have a non empty intersection with the unsafe states and we are guaranteed to find a valid counterexample.

The only way that the procedure may not terminate is if it keeps getting spurious counterexamples in an infinite sequence. At any stage of the learning process, let $(S^+, S_1^-, S_{2a}^-, S_{2b}^-)$ be the set of positive and negative examples and let L be the language learned by RPNI based on these examples. As mentioned before, RPNI constructs a prefix tree automata (PTA) which simply consist of paths leading to strings in S^+ with common prefixes merged together. L is obtained by merging states in the PTA while making sure none of the negative examples are accepted. For a given S^+, if the counterexample l_c for safety property is found to be spurious, then in the next stage RPNI is provided with $(S^+, S_1^-, S_{2a}^-, S_{2b}^- \cup \{l_c\})$. The new language L_l' is necessarily different than L_l since they do not agree on the acceptance of l_c. Since the PTA is finite, merging different states can generate only finitely many languages. In the extreme case, no state in the PTA is merged; but then there cannot be any spurious counterexample since the PTA is based on S^+ which are obtained from valid executions. Hence, the procedure cannot get "stuck" getting an infinite sequence of negative examples without any change in the traces obtained.

B Detailed description of examples

B.1 Data with parity

The system consists of a sender and a receiver whose automata are shown in Figure 4. The sender attaches a "parity" bit if the number of 1's is not even which is verified by the receiver. The FIFO system is simply a cross product of the two automata with the bad states being any state in which the receiver is at the control state q_b.

B.2 Resource arbitrator

Consider a system with two senders which share a channel c_0 for broadcasting data. In order to ensure that only one sender transmits at a time, there is a resource

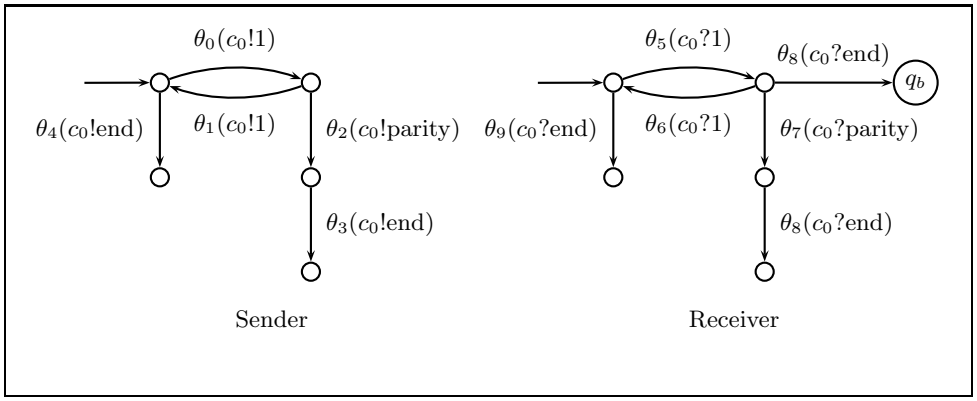

Fig. 4. FIFO automata for data transmission with parity

manager to which a request has to be sent for the use of channel c_0. The resource managers grants the request to one sender and waits for that sender to indicate that it is done before listening for more requests. For simplicity, we assume that the first sender transmits a pattern 01^* on c_0 while the second sender transmits 2^*. The safety property we want to verify that in any control state, the channel c_0 does not have transmissions from the two senders mixed up, *i.e.*, for any control state, contents of c_0 other than $((01)^*|2^*)^*$ is considered "unsafe". Figure 5 shows the description of the senders and the resource manager; the automaton to be analyzed is the cross product of all these three automata. It can be seen that $AL(F)$ for this automaton is regular.

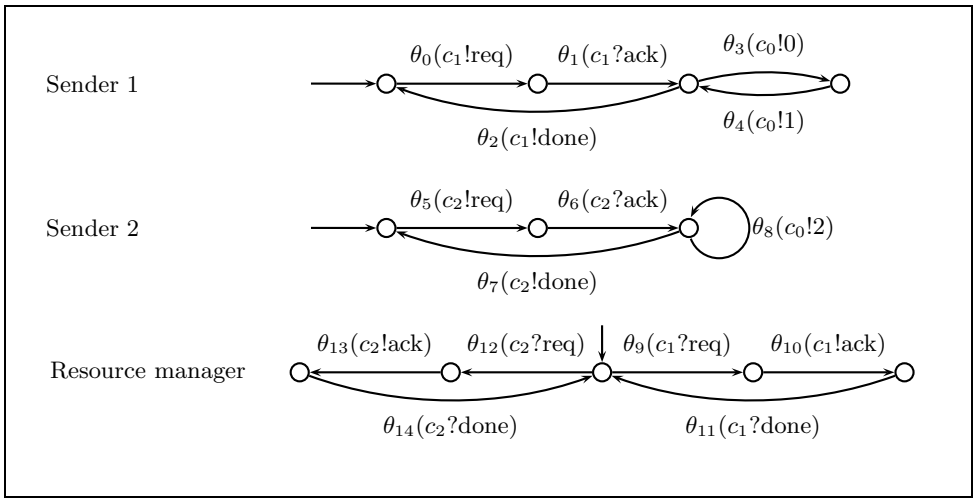

Fig. 5. FIFO automata for resource arbitrator