Actively learning to verify safety for FIFO automata

Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, Gul Agha [⋆]
Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, USA
{vardhan,ksen,vmahesh,agha}@cs.uiuc.edu

Abstract. We apply machine learning techniques to verify *safety* properties of *finite state machines* which communicate over *unbounded FIFO channels*. Instead of attempting to iteratively compute the reachable states, we use *Angluin's L* algorithm* to learn these states symbolically as a regular language. The learnt set of reachable states is then used either to prove that the system is safe, or to produce a valid execution of the system that leads to an unsafe state (*i.e.* to produce a counterexample). Specifically, we assume that we are given a model of the system and we provide a novel procedure which answers both *membership* and *equivalence* queries for a representation of the reachable states. We define a new *encoding* scheme for representing reachable states and their witness execution; this enables the learning algorithm to analyze a larger class of FIFO systems automatically than a naive encoding would allow. We show the upper bounds on the running time and space for our method. We have implemented our approach in Java, and we demonstrate its application to a few case studies.

1 Introduction

Infinite state systems often arise as natural models for various software systems at the design and modeling stage. An interesting class of infinite state systems consists of finite state machines that communicate over unbounded first-in-first-out channels, called *FIFO automata*. FIFO automata are commonly used to model various communication protocols; languages, such as 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 done by an exhaustive exploration of the state-space. However, for infinite state systems, exhaustive exploration of the state space is impossible; in fact, the verification problem in general can shown to be undecidable.

In the Lever (LEarning to VERify) project, we are pursuing the goal of using *machine learning* techniques for verification of infinite state systems. The idea

[⋆] The third author was supported in part by DARPA/AFOSR MURI Award F49620-02-1-0325 and NSF 04-29639. The other three authors were supported in part by DARPA IPTO TASK Program (contract F30602-00-2-0586), ONR Grant N00014-02-1-0715, and Motorola Grant MOTOROLA RPS #23 ANT

is as follows. Instead of computing the reachable states by iteratively applying the transition relation until a fixpoint is reached (which may not be possible in a finite number of iterations), we view the identification of the reachable states as a *language inference* problem. Naturally, in order for a learner to be able to learn the reachable region, we have to provide it with some information about the reachable states. We can easily find examples of reachable states by executing some sample sequence of transitions. Moreover, given a set of states as the supposed reachable region, we can check if this set is a fixpoint under the transition relation. If it is not a fixpoint then clearly it is not the correct reachable region. However, most learning algorithms also require either negative examples of the concept being learned or the ability to make membership and equivalence queries. To provide this information, the algorithm learns an *annotated trace language* representing reachable states as well as system executions witnessing the reachability of these states. If the learning algorithm outputs a set of traces that is closed under the transition relation of the system and does not reach any of the unsafe states then clearly the system can deemed to be correct. On the other hand, unsafe states output by the learning algorithm can be used to obtain executions (called counter-examples) leading to the unsafe state because we learn traces which provide witnesses along with the reachable states. Spurious counter-examples can be used by the learner to refine the hypothesis, and the process is repeated until either a valid counterexample is found or the system is shown to be correct. Finally, based on the practical success enjoyed by *regular model checking* [6], we assume that the set of annotated traces to be learnt is regular. Our main observation is that this learning based approach is a *complete verification* method for systems whose annotated trace language is regular (for a precise condition see Section 4). In other words, for such systems, we will eventually either find a buggy execution that violates the safety property, or we will successfully prove that no unsafe state is reachable. We have previously applied the RPNI[11] algorithm for verification of safety properties [14].

This paper presents two main new ideas. Firstly, we give a new scheme for the *annotations* on traces. With this annotation scheme, many more practical FIFO systems have regular annotated trace languages, thus enlarging the class of systems that can be provably verified by our method. Secondly and more significantly, we provide a method to devise a *knowledgeable teacher* which can answer membership (whether a string belongs to the target) as well as equivalence-queries (given a hypothesis, whether it matches the concept being learnt). In the context of learning annotated traces, equivalence queries can be answered only to a limited extent. However, we overcome our limitation to answer equivalence queries exactly and present an approach that is still able to use the powerful query-based learning framework. Our decision to use Angluin's L* algorithm [2] gives us significant benefits. First, the number of samples we need to consider is polynomial in the size of the *minimal* automaton representing the annotated traces. Second, we are guaranteed to learn the minimal automaton that represents the annotated traces. Finally, we can show that the running time is bounded by a polynomial in the size of the minimal automaton representing the annotated traces and the time taken to verify if an annotated trace is valid for the FIFO system.

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 previously proposed algorithmic verification methods; there are examples of FIFO automata that our method successfully verifies; however other approaches, fail (see [13]). We give the requirements under which classes of infinite state systems other than FIFO automata can be verified using the learning approach. Proofs of propositions and the details of the complexity analysis are available in the full version of the paper [13].

Related Work: For automatic verification of infinite state FIFO systems, the state space has to be represented by symbolic means. Some common representations are regular sets [6, 1], Queue Decision Diagrams [4], semi-linear regular expressions [7], and constrained QDDs [5]. Since an iterative approach of computing the fixpoint for reachability may not terminate, various mechanisms are used for finding the reachable set. In the approach using *meta-transitions* and *acceleration* [4, 5, 7], a sequence of transitions, referred to as a *meta-transition*, is selected, and the effect of its infinite iteration is calculated. Another popular method for verification of FIFO automata (and parameterized and integer systems) is *regular model checking* [6, 1] where reachable states are represented as regular sets, and a transducer is used to represent the transition relation. An approach for computing the reachable region that is closely related to ours is *widening* given in [6] and extended in [12] for parametric systems. However, in addition to proving a system correct, our approach can also detect bugs, which is not possible using widening (except for certain special contexts where it can be shown to be exact).

We introduced the learning to verify approach in [14], where we used RPNI [11] to learn the regular set from positive and negative queries without active queries. Concurrently and independently of our work, Habermehl *et al.* [8] have also proposed a learning based approach for verification of systems whose transition can be represented by a length-preserving transducer. They find all strings of a certain length that can be reached from the initial state and use a state merging algorithm to learn the regular set representing the reachable region.

A more detailed description of the related work is available from the full version of this paper [13].

2 Learning framework

We use Angluin's L* algorithm [2] which falls under the category of *active learning*. Angluin's L* algorithm requires a *Minimally Adequate Teacher*, which provides an oracle for membership (whether a given string belongs to a target regular set) and equivalence queries (whether a given hypothesis matches the target regular set). If the teacher answers *no* to an equivalence query, it also provides a string in the symmetric difference of the hypothesis and the target sets. The main idea behind Angluin's L* algorithm is to systematically explore strings in the alphabet for membership and create a DFA with minimum number of states to make a conjecture for the target set. If the conjecture is incorrect, the string returned by the teacher is used to make corrections, possibly after more membership queries. The algorithm maintains a prefix closed set S representing different possible states of the target DFA, a set SA for the transition function consisting of strings from S extended with one letter of the alphabet, and a suffix closed set E denoting *experiments* to distinguish between states. An *observation table* with rows from $(S \cup SA)$ and columns from E stores results of the membership queries for strings in $(S \cup SA).E$ and is used to create the DFA for a conjecture. Angluin's algorithm is guaranteed to terminate in polynomial time with the minimal DFA representing the target set.

3 FIFO Automata

A FIFO automaton [7] 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. We use the standard operational semantics of FIFO automata in which channels are considered to be perfect and messages sent by a sender are received in the order in which they were sent. For states $s_1, s_2 \in S = Q \times (M^*)^C$, we write $s_1 \xrightarrow{\theta} s_2$ if the transition θ leads from s_1 to s_2. For $\sigma = \theta_1\theta_2 \cdots \theta_n \in \Theta^*$, we say $s \xrightarrow{\sigma} s'$ when there exist states $s_1 \ldots s_{n-1}$ such that $s \xrightarrow{\theta_1} s_1 \xrightarrow{\theta_2} \cdots s_{n-1} \xrightarrow{\theta_n} s'$. The trace language of the FIFO automaton is $L(F) = \{\sigma \in \Theta^* \mid \exists s.\ s_0 \xrightarrow{\sigma} s\}$ where $s_0 = (q_0, (\epsilon, \ldots, \epsilon))$, i.e., the initial control state with no messages in the channels.

4 Verification procedure

We assume that we are given a model of the FIFO automata which enables us to identify the transition relation of the system. To use Angluin's L* algorithm for learning, we need to answer both membership and equivalence queries for the reachable set. However, there is no immediate way of answering a membership query (whether a certain state is actually reachable or not). Therefore, instead of learning the set of reachable states directly, we learn a language which allows us to identify both the reachable states and candidate witnesses (in terms of the transitions of the system) to these states. The validity of any witness can then be checked, allowing membership queries to be answered.

For equivalence queries, we can provide an answer in one direction. We will show that the reachable region with its witness executions can be seen as the least fixpoint of a relation derived from the transitions. Hence, an answer to the equivalence query can come from checking if the proposed language is a fixpoint under this relation. If it is not a fixpoint then it is certainly not equivalent to the target; but if it is a fixpoint, we are unable to tell if it is also the least fixed point. However, we are ultimately interested in only checking whether a given safety property holds. If the proposed language is a fixpoint but does not intersect with the unsafe region, the safety property clearly holds and we are done. On the other hand, if the fixpoint does intersect with unsafe states, we can check if such an unsafe state is indeed reachable using the membership query. If the unsafe state is reachable then we have found a valid counterexample to the safety property and are done. Otherwise the proposed language is not the right one since it contains an invalid trace.

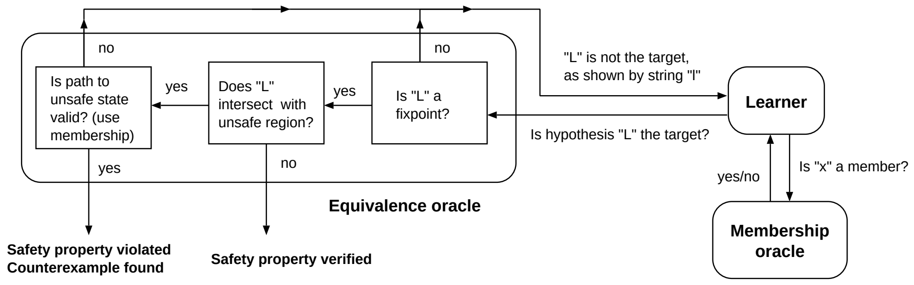

Fig. 1. Verification procedure

Figure 1 shows the high level view of the verification procedure. The main problems we have to address now are:

- What is a suitable representation for the reachable states and their witnesses?
- Given a language representation, we need to answer the following questions raised in Figure 1:
  - (Membership Query) Given a string x, is x a valid string for a reachable state and its witness?
  - (Equivalence Query(I)) Is a hypothetical language L a fixpoint under the transition relation? If not, we need a string which demonstrates that L is not a fixpoint.
  - (Equivalence Query(II)) Does any string in L witness the reachability of some “unsafe” state?

4.1 Representation of the reachable states and their witnesses

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.[1] 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.

In [14], we overcame this difficulty by annotating the traces in a way that makes it possible to compute the set of reachable states. We briefly describe this annotation scheme before presenting the actual scheme used in this paper. Consider a set $\overline{\Theta}$ of *co-names* defined as follows:

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

Thus, for every send or receive action in our FIFO automaton, there is a new transition name with a *bar*. A *barred* transition $\overline{\theta}$ in an annotated trace of the system denotes either a message sent that will later be consumed, or the receipt of a message that was sent earlier in the trace. Annotated traces of the automaton are obtained by marking send-receive pairs in a trace exhibited by the system.

The above annotation scheme allowed us to calculate the reachable set for any regular set of annotated traces by a simple homomorphism. However, one difficulty we encountered is that for some practical FIFO systems, the annotated trace language is not regular; the nonregularity often came from the fact that a receive transition has to be matched to a send which could have happened at an arbitrary time earlier in the past. To alleviate this problem, we use a new annotation scheme in which only the send part of the send-receive pair is kept. This gives an annotated trace language which is regular for a much larger class of FIFO systems (although we cannot hope to be able to cover all classes of FIFO systems since they are Turing expressive). We now describe this annotation in detail.

As before, we have a new set of barred names but this time only for the send transitions:

$$\overline{\Theta} = \{\overline{\theta} \mid \theta \in \Theta \text{ and } \delta(\theta) \in Q \times \{c_i!a_j\} \times Q \text{ for some } c_i, a_j\}$$

We also define another set of names $T_Q = \{t_{\mathrm{q}} \mid q \in Q\}$ consisting of a symbol for each control state in the FIFO.

Now let the alphabet of *annotated traces* Σ be defined as $(\Theta - \Theta_r) \cup \overline{\Theta} \cup T_Q$ where Θ_r is the set of receive transitions $\{\theta_r \mid \delta(\theta_r) \in Q \times \{c_i?a_j\} \times Q \text{ for some } c_i, a_j\ \}$.

Given a sequence of transitions l in $L(F)$, let $\mathcal{A}$ be a function which produces an annotated string in Σ^*. $\mathcal{A}$ takes each receive transition θ_{r_i} in l and finds the matching send transition θ_{s_i} which must occur earlier in l. Then, θ_{r_i} is removed and θ_{s_i} replaced by $\overline{\theta_{s_i}}$. Once all the receive transitions have been accounted

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

for, $\mathcal{A}$ appends the symbol $t_q \in T_Q$ corresponding to the control state q which is the destination of the last transition in l. Intuitively, for a send-receive pair which cancel each other's effect on the channel contents, $\mathcal{A}$ deletes the received transition and replaces the send transition with a barred symbol. As before, a barred symbol indicates that the message sent gets consumed by a later receive. Notice that in the old annotation scheme both the send and the receive were replaced with a barred version; here the receive transition is dropped altogether. The reason we still keep the send transition with a bar is, as we will show shortly, that this allows us to decide whether any given string is a valid annotated trace. The symbol t_q is appended to the annotated trace to record the fact that the trace l leads to the control state q.

As an example, consider the FIFO automaton shown in Figure 2. For the following traces in $L(F)$: $\theta_1\theta_2\theta_3$, $\theta_1\theta_2\theta_3\theta_1\theta_2$, the strings output by $\mathcal{A}$ are respectively: $\overline{\theta_1}\theta_3 t_{q_0}$, $\overline{\theta_1\theta_3}\theta_1 t_{q_2}$.

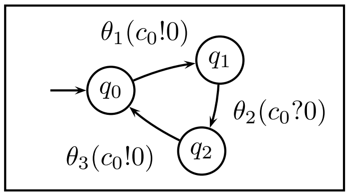

Fig. 2. A FIFO automaton

Let the language of annotated traces be $AL(F) = \{\mathcal{A}(t) \mid t \in L(F)\}$ which consists of all strings in Σ^* that denote correctly annotated traces of F. Let $AL^{\text{old}}(F)$ be the annotated trace language corresponding to the old annotation scheme described earlier (in which we keep both parts of a send-receive pair). The following proposition shows that the new annotation scheme has regular annotated trace language for more FIFO automata than the old scheme.

Proposition 1. *The set of FIFO automata for which $AL(F)$ is regular is strictly larger than the set of FIFO automata for which $AL^{\text{old}}(F)$ is regular.*

$AL(F)$ can be seen to represent both the reachable states of the FIFO system and the annotated traces which in some sense witness the reachability of these states. Thus, $AL(F)$ is a suitable candidate for the language to use in the verification procedure shown in Figure 1.

Given a string l in Σ^*, we say that l is well-formed if l ends with a symbol from T_Q and there is no other occurrence of symbols from T_Q. We say that a language L is well-formed if all strings in L are well-formed. For a well-formed string l ending in symbol t_q, let $\mathcal{T}(l)$ denote the prefix of l without t_q and let $\mathcal{C}(l)$ denote the control state q.

4.2 Answering membership queries

In order to answer a membership query for $AL(F)$, given a string l in Σ^* we need to verify if l is a correct annotation for some valid sequence of transitions l' in $L(F)$. Let $\mathcal{A}^{-1}(l)$ be a function which gives the set (possibly empty) of all sequences of transitions l' for which $\mathcal{A}(l') = l$. First, if l is not well-formed, $\mathcal{A}^{-1}(l) = \emptyset$ since all valid annotations are clearly well-formed. Assuming l is well-formed, if we ignore the bars in $\mathcal{T}(l)$, we get a string l'' which could potentially be in $\mathcal{A}^{-1}(l)$ except that the transitions corresponding to any receives are missing. We can identify the possible missing receive transitions by looking at the barred symbols in $\mathcal{T}(l)$; each barred send can potentially be matched by a receive transition that operates on the same channel and has the same letter. However, we do not know the exact positions where these receive transitions are to be inserted in l''. We can try all possible (finitely many) positions and simulate each resulting transition sequence on the fly on the FIFO system. Any transition sequence which is valid on the FIFO and gives back l on application of $\mathcal{A}$ is then a member of $\mathcal{A}^{-1}(l)$. If $\mathcal{A}^{-1}(l) \neq \emptyset$ then l is a valid annotated trace.

For illustration, let us consider a membership query for the string $\overline{\theta_1 \theta_3} \theta_1 t_{q_2}$ for the FIFO automata shown in Figure 2. We identify the possible missing receive transitions as two instances of θ_2. Since a receive can only occur after a send for the same channel and letter, the possible completions of the input string with receives are $\{\theta_1\theta_2\theta_3\theta_2\theta_1, \theta_1\theta_2\theta_3\theta_1\theta_2, \theta_1\theta_3\theta_2\theta_2\theta_1, \theta_1\theta_3\theta_2\theta_1\theta_2, \theta_1\theta_3\theta_1\theta_2\theta_2\}$. Of these, $\theta_1\theta_2\theta_3\theta_1\theta_2$ can be correctly simulated on the FIFO system and gives back the input string $\overline{\theta_1 \theta_3} \theta_1 t_{q_2}$ on application of $\mathcal{A}$. Therefore, the answer to the membership query is *yes*. An example for a negative answer is $\overline{\theta_1} t_{q_0}$.

4.3 Answering equivalence queries

For learning $AL(F)$ in the active learning framework, we need a method to verify whether a supposed language L of annotated traces is equivalent to $AL(F)$. If not, then we also need to identify a string in the symmetric difference of $AL(F)$ and L to allow the learner to make progress.

Given a string $l \in L$ and a transition θ in the FIFO, we can find if it is possible to *extend* l using θ. More precisely, we define a function $Post(l, \theta)$ as follows. If l is well-formed, let $source(\theta)$ and $target(\theta)$ be the control states which are respectively the source and the target of θ.

$$Post(l,\theta) = \begin{cases} \emptyset & \text{if } l \text{ not well-formed or if } \mathcal{C}(l) \neq source(\theta) \\ \{\mathcal{T}(l)\theta \, t_{\text{target}(\theta)}\} & \text{otherwise if } \delta(\theta) = \tau \text{ or } \delta(\theta) = c_i!a_j \\ \{deriv(\mathcal{T}(l), \theta) \, t_{\text{target}(\theta)}\} & \text{otherwise if } \delta(\theta) = c_i?a_j \end{cases}$$

$deriv(\mathcal{T}(l), \theta)$ checks the first occurrence of a send θ_s in $\mathcal{T}(l)$ for channel c_i and if the send is for the character a_j, replaces θ_s with $\overline{\theta_s}$. $deriv(\mathcal{T}(l), \theta)$ is empty if no such θ_s could be found or if θ_s outputs a character other than a_j. Intuitively, *deriv* is similar to the concept of the derivative in formal language theory, except that we look at only the channel that θ operates upon.

Let $Post(l)$ be $\bigcup_{\theta \in \Theta} Post(l, \theta)$ and $Post(L)$ be $\bigcup_{l \in L} Post(l)$.

Theorem 1. *Let $\mathcal{F}(L) = Post(L) \cup \{t_{q_0}\}$ where q_0 is the initial control state. $\mathcal{F}(L)$ is a monotone set operator,* i.e. *it preserves set-inclusion. Moreover, $AL(F)$ is the least fixpoint of the functional $\mathcal{F}(L)$.*

Theorem 1 gives us a method for answering equivalence queries for $AL(F)$ in one direction. If L is not a fixpoint, it cannot be equivalent to $AL(F)$. In this case, we can also find a string in $L \oplus AL(F)$ as required for Angluin's algorithm. Here, $A \oplus B$ denotes the symmetric difference of two sets. Consider the following cases:

1. $\mathcal{F}(L) - L \neq \emptyset$. Let l be some string in this set. If l is t_{q_0} then it is in $AL(F) \oplus L$. Otherwise, we can check if l is a valid annotation using the procedure described in Section 4.2. If yes, then l is in $AL(F) \oplus L$. Otherwise, it must be true that $l \in Post(l')$ for some $l' \in L$. If l is not valid, l' cannot be valid since $Post()$ of a valid annotation is always valid. Hence $l' \notin AL(F)$ or $l' \in AL(F) \oplus L$.
2. $\mathcal{F}(L) \subsetneq L$. From standard fixpoint theory, since $AL(F)$ is the least fixed point under $\mathcal{F}$, it must be the intersection of all prefixpoints of $\mathcal{F}$ (a set Z is a prefixpoint if it *shrinks* under the functional $\mathcal{F}$, *i.e.* $\mathcal{F}(Z) \subseteq Z$). Now, L is clearly a prefixpoint. Applying $\mathcal{F}$ to both sides of the equation $\mathcal{F}(L) \subsetneq L$ and using monotonicity of $\mathcal{F}$, we get $\mathcal{F}(\mathcal{F}(L)) \subsetneq \mathcal{F}(L)$. Thus, $\mathcal{F}(L)$ is also a prefixpoint. Let l be some string in the set $L - \mathcal{F}(L)$. Since l is outside the intersection of two prefixpoints, it is not in the least fixpoint $AL(F)$. Hence, l is in $AL(F) \oplus L$.
3. $\mathcal{F}(L) = L$. Let $\mathcal{W}(L)$ be the set of annotated traces in L which can reach unsafe states (We will describe how $\mathcal{W}(L)$ is computed in the next section). If $\mathcal{W}(L)$ is empty, since L is a fixpoint, we can abort the learning procedure and declare that the safety property holds. For the other case, if $\mathcal{W}(L)$ is not empty then let l be some annotated trace in this set. We check if l is a valid annotation using the procedure described in Section 4.2. If it is valid, we have found a valid counterexample and can again abort the whole learning procedure since we have found an answer (in the negative) to the safety property verification. Otherwise, l is in $AL(F) \oplus L$.

A subtle point to note is that although we attempt to learn $AL(F)$, because of the limitation in the equivalence query, the final language obtained after the termination of the verification procedure may not be $AL(F)$. It might be some fixpoint which contains $AL(F)$ or it might be simply some set which contains a valid annotated trace demonstrating the reachability of some unsafe state. However, this is not a cause for concern to us since in all cases the answer for the safety property verification is correct.

4.4 Finding annotated traces leading to unsafe states

In the previous section, we referred to a set $\mathcal{W}(L)$ in L which can reach unsafe states. We now show how this can be computed.

We assume that for each control state $q \in Q$, we are given a recognizable set [3] describing the unsafe channel configurations. Equivalently, for each q, the unsafe channel contents are given by a finite union of products of regular languages: $\bigcup_{0 \leq i \leq n_q} P_{q,i}$ where $P_{q,i} = \prod_{0 \leq j \leq k} U_q(i, c_j)$ and $U_q(i, c_j)$ is a regular language for contents of channel c_j. For each $P_{q,i}$, an unsafe state s_u is some $(q, u_0, u_1, \ldots u_k)$ such that $u_j \in U_q(i, c_j)$.

For a channel c, consider a function $h_c : \Sigma \rightarrow M^*$ 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 Σ^* to M^* that extends the above function.

Let L_q be the subset of an annotated trace set L consisting of all well-formed strings ending in t_q, *i.e.* $L_q = \{l \mid l \in L \text{ and } \mathcal{C}(l) = q\}$.

If an unsafe state $s_u = (q, u_0, u_1, \ldots u_k)$ is reachable, then there must exist a sequence of transitions $l_\theta \in \Theta^*$ such that $s_0 \xrightarrow{l_\theta} s_u$, where s_0 is the initial state. In l_θ, if the receives and the sends which match the receives are taken out, only the remaining transitions which are sends can contribute to the channel contents in s_u. Looking at the definition of h_c, it can be seen that for each channel content u_j in s_u, $u_j = h_{c_j}(\mathcal{A}(l_\theta))$ (recall that $\mathcal{A}$ converts a sequence of transitions into an annotated trace). Thus, for s_u to be reachable, there must be some annotated trace $l \in AL(F)$ such that $s_u = (\mathcal{C}(l), h_{c_0}(l), h_{c_1}(l), \ldots, h_{c_k}(l))$.

Let $h_{c_j}^{-1}(U_q(i, c_j))$ denote the inverse homomorphism of $U_q(i, c_j)$ under h_{c_j}. For each $P_{q,i}$, $\bigcap_{0 \le j \le k} h_{c_j}^{-1}(U_q(i, c_j))$ gives a set of annotated strings which can reach the unsafe channel configurations for control state q. Intersecting this with L_q verifies if any string in L can reach these set of unsafe states. If we perform such checks for all control states for all $P_{q,i}$, we can verify if any unsafe state is reached by L. Thus, the set of annotated traces in L that can lead to an unsafe state is given by:

$$\mathcal{W}(L) = \bigcup_{q \in Q} (\bigcup_{0 \le i \le n_q} (L_q \cap \bigcap_{0 \le j \le k} h_{c_j}^{-1}(U_q(i, c_j))))$$

We summarize the verification algorithm in Figure 3.

Theorem 2. *For verifying safety properties of FIFO automata, the learning to verify algorithm satisfies the following properties:*

1. *If an answer is returned by algorithm, it is always correct.*
2. *If $AL(F)$ is regular, the procedure is guaranteed to terminate.*
3. *The number of membership and equivalence queries are at most as many as needed by Angluin's algorithm. The total time taken is bounded by a polynomial in the size of the minimal automaton for $AL(F)$ and linear in the time taken for membership queries for $AL(F)$.*

5 Generalization to other infinite state systems

The verification procedure described for FIFO automata can be generalized to other infinite state systems. The challenge for each class of system is to identify the alphabet Σ which provides an annotation enabling the following:

– membership query for the annotated trace language,
– function to compute $Post()$ for a given annotated set, and
– function to find if a string in an annotated set can reach an unsafe state

Notice that in the verification procedure we do not assume anything else about FIFO automata other than the above functions. In fact, the learning algorithm does not have to be limited to regular languages; any suitable class of languages can be used if the required decision procedures are available.

```
algorithm learner
begin
Angluin's $L^*$ algorithm
end
```

```
algorithm isMember
Input: Annotated trace $l$
Output: is $l \in AL(F)$?
begin
  if $l$ not well-formed return no
  else
    find receives matching barred symbols
    find possible positions for receives
    simulate resulting strings on FIFO
      system on the fly
    if any string reaches $\mathcal{C}(l)$ with
      correct annotation, return yes
  return no
end
```

```
algorithm Equivalence Check
Input: Annotated trace set $L$
Output: is $L = AL(F)$?
If not, then some string in $L \oplus AL(F)$
begin
  $\mathcal{F}(L) = Post(L) \cup \{t_{q_0}\}$
  if $\exists l \in (\mathcal{F}(L) - L)$
    if isMember(l)
      return (no, $l$)
    else
      return (no, $l'$ where $l = Post(l')$)
  else if $\mathcal{F}(L) \subsetneq L$
    return (no, $l \in (L - \mathcal{F}(L))$)
  else if $\exists l \in \mathcal{W}(L)$
    if isMember(l)
      Print (safety prop. does not hold, $l$); stop
    else
      return (no, $l$)
  else
    Print (safety prop. holds); stop
end
```

Fig. 3. Learning to verify algorithm

6 Implementation

We have updated the LEVER (LEarning to VERify) tool suite first introduced in [14] with the active learning based verification procedure for FIFO automata. The tool, written in Java, is available from [9]. We use a Java DFA package available from `http://www.brics.dk/~amoeller/automaton/`.

We have used LEVER to analyze some canonical FIFO automata verification problems: *Producer Consumer, Alternating bit protocol* and *Sliding window protocol* (window size and maximum sequence number 2). Table 1 shows the results obtained. We compare the number of states of the final automaton (Size) and the running times (T) using the verification procedure in this paper with the procedure we used earlier in [14] (columns Size$_{\text{old}}$ and T_{old}). It can be seen that there is an improvement using the new procedure (although the comparison of Size should be taken with the caveat that the annotation in the two procedures is slightly different). All executions were done on a 1594 MHz notebook computer with 512 MB of RAM using Java virtual machine version 1.4.1 from Sun Microsystems. We also report the time taken (T_{rmc}) by the regular model checking tool [10] on the same examples. Although a complete comparative analysis with all available tools remains to be done, it can be seen

	Size	T	Size$_{\text{old}}$	T_{old}	T_{rmc}
Producer Consumer	7	0.3s	20	0.4s	3.3s
Alternating Bit	33	2s	104	4.1s	24.7s
Sliding Window	133	54s	665	81.2s	78.4s

Table 1. Running time

the running time of Lever is slightly better than the regular model checking 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. D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, Nov. 1987.
3. J. Berstel. *Transductions and Context-Free-Languages.* B.G. Teubner, Stuttgart, 1979.
4. 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.
5. 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.
6. 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.
7. A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1–31, 2003.
8. P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In *Proc. of Infinity'04, London, UK (to appear)*, 2004.
9. LEVER. Learning to verify tool. http://osl.cs.uiuc.edu/~vardhan/lever.html, 2004.
10. M. Nilsson. http://www.regularmodelchecking.com, 2004.
11. 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.
12. T. Touili. Regular model checking using widening techniques. In *ENTCS*, volume 50. Elsevier, 2001.
13. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata (full version). http://osl.cs.uiuc.edu/docs/lever-active/activeFifo.pdf, 2004.
14. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *Proc. of ICFEM'04, Seattle, USA (to appear)*, 2004.