Improved Probabilistic Verification of Systems with a Large State Space

Shou-pon Lin
Department of Electrical Engineering
Columbia University
New York, USA
shouponlin@ee.columbia.edu

Nicholas F. Maxemchuk
Department of Electrical Engineering
Columbia University
New York, USA
nick@ee.columbia.edu

Abstract—**The probabilistic verification technique presented in this paper computes a bound for error probability, even when the state space is too large to be traversed completely. Very large state spaces occur in multi-party protocols that are needed for collaborative intelligent vehicles. Most other verification techniques are not useful because they cannot generate all of the reachable states and determine the correctness of a system. Probabilistic verification explores the states that are more likely to be encountered first, then the states that are less probable. When being constrained on memory, it stops state traversal and computes a bound for the probability of reaching the unexplored states. The bound is obtained by solving a linear program. This bound is significantly tighter than the bound that is obtained using earlier implementations of probabilistic verification. For comparison, we apply probabilistic verification and PRISM to an 7-party lock protocol used in a collaborative driving application. Probabilistic verification is successful and produces a probability bound, while PRISM cannot complete the verification because of the large composite state space of the 7-party protocol.**

I. Introduction

Verification techniques, or model checking techniques, are used to evaluate the correctness of computer communication protocols or embedded system. These systems often have a small number of concurrent processes resulting in a composite state space that is tractable for most verification techniques. With the advent of cyber-physical systems, such as collaborative driving applications, systems with more concurrent processes have to be considered. Increasing the number of collaborating processes increases the composite state space beyond the capabilities of model checkers that must test every state before completing the verification.

There are developments that improve the scalability of model checking techniques. The following strategies attempt to reduce either the problem size or the storage required to store the model of the system. Symmetry reduction [1] and partial order reduction [2], [3] are techniques that reduce the size of the model. Abstraction refinement [4], [5] is yet another class of techniques that create a less precise, but more tractable, model. Symbolic model checking techniques [6] use data structures based on binary decision diagrams (BDDs) to provide efficient storage and manipulation of the full model.

Another strategy is to come up with bounding techniques that explore a subset of the reachable states, and provide diagnosis based on the partial exploration. The symbolic method of Bounded-model checking [7], [8] explores execution steps up to a bounded length. There are also depth-bounding techniques for explicit state model checking [9] which explore the states that are reachable within a given number of steps from the initial states. They are primarily used for bug detection, but are unable to prove safety for systems with a large state space.

This paper discusses *probabilistic verification* that also explores a subset of the states in the model. It explores the states that are more likely to be reached from the initial states first. If the state space is too large to be verified completely, the unexplored states are the ones that are less likely to be encountered. It then computes a bound on the error probability. This verification technique is based on the concept of skewed-probabilistic transitions from [10], which is particularly suitable for verifying systems that interface with the physical world. Many systems, such as error recovery protocols, have transitions that occur most often, e.g. correct reception, and transitions that occur infrequently, e.g. lost messages. We labels these events differently according to the likelihood that they occur, rather than specifying their exact probabilities. The most likely events are labeled as high probability events, while less likely events are labeled as level-1 low probability events, events that are even rarer are labeled as level-2 low probability events, and so forth.

Multiple levels of low probability events are particularly useful when we verify protocols in a layered architecture, such as the architecture that is proposed for intelligent driving applications [11]. We verify a protocol dependent on the services provided by a lower layer protocol. That protocol may not have been completely verified, but may tolerate several failures of physical devices. The current protocol must be verified for both failures in physical devices and failures of the underlying protocol, which have different levels of probability of failure. We can then bound the probability of an execution sequence depending on the probability bounds of the low probability events within such execution sequence.

Probabilistic verification consists of a verification algorithm and a method to compute the probability bound for reaching an unexplored state. First, it uses the verification algorithm to explore the reachable states, with the states that are more likely to occur being examined first, until it runs out of computing resources. If the verification algorithm completely traverses the state space without depleting the memory, the result it produces is the same as any explicit-state model checkers. If the verification algorithm is unable to completely examine all the reachable states, a probability bound for the system to encounter any of the unexplored states is computed by solving a linear program, which produces a significantly tighter (or, in other word, more accurate) probability bound on the system's correctness than its predecessor [10].

This paper is organized in the following order. Section II describes the models for the transition systems to be verified, and the structural changes of the reachable graph and the execution sequences. In section III, we first briefly review the depth-first search algorithm, then we present the verification algorithm and its proof of correctness. The linear program that yields tha probability bound is derived in section IV. We apply probabilistic verification to a protocol that involves multiple concurrent processes, and the result is presented in section V. We discuss some future work and conclude the paper in section VI.

II. Preliminaries

A transition system is a collection of concurrent processes or components. These concurrent processes can either execute asynchronously or interact through synchronizing messages. Markov decision processes (MDPs) are used to model a wide range of transition systems that exhibit deterministic behaviors, probabilistic behaviors and non-deterministic behavior [12]. Probabilistic behaviors are those events associated with some probability distribution, such as message losses during communication. Non-deterministic behaviors can represent competing actions of concurrent processes, such as initiating a sequence of operations. In the lock protocol different sets of participants may request a lock. Each state in MDP represents a possible configuration of the system, and each transition in the MDP represents an execution step by one of the processes or several processes that execute synchronously.

This section describes a variation of MDPs, in which the probability choices are not assigned exact probabilities, but are labeled as high probability choices and low probability choices. For instance, in communication protocol, most of the time messages are received correctly, which are high probability events; but we must also consider infrequent messages losses, which are low probability events. Events that are labeled as high probability are much common than the events labeled as low probability. That is, the probability distributions are skewed. There are even different levels of low probability choices: a message loss on a channel can be infrequent, while failure of error recovery protocol is even rarer.

Instead of calculating an exact probability of occurrence for a sequence of states and transitions, we calculate an upper bound on the probability, dependent on the number and type of unlikely transitions in the sequence and the upper bounds on their probability of occurence. The skewed probabilities lead to equivalence classes for reachable states. Finally, we define the errors that the verification algorithm checks.

A. The model

Definition 1: A skewed-probability Markov decision process is a tuple $M = (S, A, \hat{p}, \pi, I)$, where

- S is a countable set of states
- A is a set of non-deterministic actions
- $\hat{p}$ is the maximum value of the probability of infrequent events
- $\pi : S \times A \times S \rightarrow \mathbb{Z}^+$ is a labeling function such that for all state s and action α,
  - $\pi(s, \alpha, s') = i$ implies that (s, α, s') has a non-zero transition probability $p_{(s,\alpha,s')} < \hat{p}^i$.
  - $\pi(s, \alpha, s')$ is undefined if $p_{(s,\alpha,s')} = 0$.
- I is the set of initial states.

An action α is said to be *enabled* in s iff $p_{(s,\alpha,s')} > 0$ for some $s' \in S$ (or equivalently, $\pi(s, \alpha, s')$ is defined for some $s' \in S$), otherwise α is *disabled*; $A(s)$ denotes the set of enabled actions in s. The α-*successors* of s are the set of states $\alpha\text{-}succ(s) = \{s' | p_{(s,\alpha,s')} > 0\}$, and the corresponding α-*transitions* from s are the set of transitions $\alpha\text{-}trans(s) = \{(s, \alpha, s') | p_{(s,\alpha,s')} > 0\}$. The transitions from s is $trans(s) = \bigcup_{\alpha \in A(s)} \alpha\text{-}trans(s)$. A transition (s, α, s') is *deterministic* if it is the only element in $trans(s)$.

A transition system modeled by MDP operates as follows. Initially, the transition system may start in some initial state s_0 in I. A transition from s is divided into two steps: a non-deterministic choice α is first selected from the set $A(s)$, then a probabilistic choice is made by selecting a successor state from $\alpha\text{-}succ(s)$.

The name skewed-probability comes from the fact that we categorize probability choices into those that correspond to the more frequent events and those that stand for the less common events. The high probability choices lead to transitions with $\pi(.) = 0$. Those leading to transitions with $\pi(.) > 0$ are low probability choices, and there can be multiple levels of low probability choices. Different levels of low probability choices differ greatly in their frequencies, too. A level-1 low probability choice that leads to a transition with $\pi(.) = 1$ occurs much more often than a level-2 low probability choice leading to a transition with $\pi(.) = 2$, and so forth.

Low probability probabilistic choices can often use to model message loss in communication systems, sensor inaccuracies or mechanical malfunction in intelligent vehicles.

The non-deterministic choices often arise from the concurrent execution of processes in a system. In such case the notion of adversaries [13] is introduced to resolve the non-determinism in a model. An adversary can represent one possible scheduling policy of the concurrent processes over the lifetime of the system. In this work, we do not rely on adversary to assign non-deterministic choices with probability. We assume that every non-deterministic choice can be selected and verify every path that follows.

B. Execution sequence and structure of reachable graph

An execution of the transition system corresponds to a sequence of states and transitions in the Markov decision process. Specifically, *execution sequences* are sequences which originate at some initial state $s_0 \in I$, and *paths* denote those sequences that are not originated at initial state. An execution sequence can be either finite or infinite (because there may be cycles in the transition system). We start by considering finite execution sequences in the transition system, and we shall see later a desirable property of a system is that a system will not stay in a cycle indefinitely.

Consider a finite sequence $\sigma = s_0\alpha_0 s_1\alpha_1 s_2 \dots \alpha_{n-1}s_n$, where s_i's are states and α_i's are enabled actions at respective s_i's. We omitted the action from the representation of σ wherever they are not relevant to the context. We use σ_0 to denote the first state in σ and σ_{-1} to denote the last state. To characterize the probability of a given sequence, we use the following definition.

Definition 2: $L(\sigma)$ is the sum of the labels $\pi(.)$ of all transitions in σ,

$$L(\sigma) = \sum_{(u,\alpha,v)\in\sigma} \pi(u,\alpha,v) \tag{1}$$

Like paths in Markov chain, a sequence in skewed-probability MDPs can be associated with some probability. Although we cannot find the exact probability of each possible sequence without exact probability distribution, a sequence $\sigma = s_0\alpha_0 s_1\alpha_1 s_2 \dots \alpha_{n-1}s_n$ can be associated with a probability bound: $P(\sigma) = p_{(s_0,\alpha_0,s_1)} p_{(s_1,\alpha_1,s_2)} \cdots p_{(s_{n-1},\alpha_{n-1},s_n)} \le \hat{p}^{L(\sigma)}$

A sequence may contain zero or several low probability transitions. If the transitions in this sequence are all high probability transitions, that is, $L(\sigma) = 0$, we refer to it as *high probability sequence*. It has a probability bound $P(\sigma) \le \hat{p}^{L(\sigma)} = 1$. If a sequence $\sigma = s_0\alpha_0 s_1\alpha_1 s_2 \dots \alpha_{n-1}s_n$ that contains three low probability transitions, specifically $\pi(s_0,\alpha_0,s_1) = \pi(s_2,\alpha_2,s3) = 2$, and $\pi(s_{n-1},\alpha_{n-1},s_n) = 1$, has $L(\sigma) = 5$ and $P(\sigma) \le \hat{p}^{L(\sigma)} = \hat{p}^5$.

The labeling function π transforms the structure of the reachable graph into layers, or equivalent classes. We can now define equivalence classes in the transition system.

Definition 3: A state s belongs to $\mathcal{C}_k$ (equivalence class k) if s is reachable from one of the initial states via some execution sequence σ s.t. $L(\sigma) = k$, and there does not exist any execution sequence σ from any of the initial states to s with $L(\sigma) < k$.

Intuitively speaking, given a state s in equivalence class $\mathcal{C}_k$, there can be multiple execution sequences that reach s. The execution sequence which is most likely to be taken to reach state s has a probability no greater than $\hat{p}^k$.

The equivalence classes have the following properties:

Property 1: $\mathcal{C}_i \cap \mathcal{C}_j = \phi,\ \forall\ i \neq j$

Property 2: $\bigcup_{i\ge 0} \mathcal{C}_i =$ All the reachable states

The two properties imply that we can explore equivalence classes in the increasing order and eventually we traverse all the reachable states. They motivate the verification algorithm.

C. Correctness properties

Definition 4: A state s is considered a deadlock if $A(s) = \phi$, i.e., there is no enabled action at state s.

If this is the case, then state s has no successor. A deadlock is an undesirable system state. The system cannot proceed further once it arrives at a deadlock.

Definition 5: A cycle $\sigma = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \dots s_n\alpha_n s_0$ is a livelock if all the transitions are deterministic, and any of $s_0, \dots, s_n$ is not an initial state.

A livelock is a loop in which the system does not perform useful work, and it is consider a system failure. The system stays in this loop forever. A system has infinite sequence if it has a livelock. In this paper, we extend the definition of livelock to include those cycles that consist of all high probability transitions, or *high probability cycles*.

Definition 6: A cycle $\sigma = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \dots s_n\alpha_n s_0$ is a livelock if all the transitions are high probability transitions, that is, $L(\sigma) = 0$, and any of $s_0, \dots, s_n$ is not an initial state.

Starting at state s_0, the probability of traversing the cycle and returning to s_0 is upper-bounded by $\hat{p}^{L(\sigma)} = 1$. Systems that have high probability loops tend to have a large average number of steps in their execution sequences without performing useful work and lead to inefficient protocols.

For a system that do not have high probability cycle, all of its cycle (if any) must contain some low probability transition. The probability of traversing a cycle containing low probability transitions and returning to the starting point must be less than $\hat{p}$. The system can only stay in such cycle indefinitely with negligible probability. A desirable property of a system is not to possess high probability cycle, for which we do not have to consider infinite execution sequences.

III. Verification Algorithm

The probabilistic verification technique presented in this paper is a composition of depth-first search and probabilistic search. Given a transition system modeled as a graph, or any models having a graph-like structure, the standard depth-first search algorithm dutifully traverses all the states that are reachable from the initial state. On the other hand, the probabilistic search [10] examines the sequences that are most likely to occur first.

We combine the two paradigms by modifying depth-first search so that it traverses all the states that are reachable without taking a low probability transition. The states following the low probability transition are stored separately in several hash table. The states that are reached with fewer low probability transitions are stored in a higher hash table. The probabilistic verification algorithm repeatedly pops entries out from the highest hash tables. It then starts depth-first search from such entry and traverses the states that are reachable without taking a low probability transition. If the verification algorithm terminates when there are no entries left in the hash table, then the transition system is completely examined. If the verification cannot complete, we use the method described in the next section to estimate the likelihood of an execution sequence leading to an unexplored state.

In this section, we briefly review the depth-first search implemented by most explicit-state model checkers. Then we describe how to integrate probabilistic search into depth-first search. Finally, we prove that the algorithm explores states in equivalence classes in ascending orders and that it reports error whenever one is found.

Algorithm 1 Depth-first search

```
procedure DFS($M$)
    for all $s \in I$ do
        if $s \notin Explored$ then
            Insert $s$ into $Explored$
            DFS-visit($M, s$)
        end if
    end for
end procedure
procedure DFS-VISIT($M, s$)
    if $A(s) = \phi$ or $s$ violates some properties then
        Report error
    end if
    for all $(s, \alpha, t) \in trans(s)$ do
        if $t \notin Explored$ then
            Insert $t$ into $Explored$
            DFS-search($M, t$)
        end if
    end for
end procedure
```

A. Depth-first search

Probabilistic verification uses a modification of depth-first search (DFS) algorithm. Algorithm 1 describes an implementation of DFS, which we will modify later on. DFS is applicable to models that can be represented as graph. The skewed-probability MDP model we use is such. DFS does not distinguish different types of transitions that appear in MDPs. It simply considers all the transitions to be on par. It visits every state that is reachable from any of the initial states. It stores all the states that have been examined in the container *Explored*, which can be implemented as a hash table.

It starts the subroutine DFS-visit from each initial state (line 2-7). For each state s, it first checks if s is a deadlock or represents an error state (line 10). It then proceeds to examine every transitions in $trans(s)$ (line 13). If a successor state t is not yet traversed, i.e., not stored in *Explored*, DFS-visit is called recursively to continue exploring the successors of t (line 16).

Since DFS traverses all reachable states, we can also extend it to check assertion of system invariants or regular safety properties. Checking regular safety properties against a transition system amounts to taking the product of the transition system and a finite automaton that embodies the regular safety property to be checked [12]. The regular safety property is deemed violated if the system can reach an acceptance state of the product of the the transition system and the finite automaton. Building such finite automaton from a regular safety property is beyond the scope of this paper.

B. Integrate probabilistic search into DFS

The modified DFS that will be described here is a result of introducing probabilistic search into typical DFS. It explores states in the decreasing order of how likely they appear in a system. Namely, the algorithm attempts to examine all states

Algorithm 2 Probabilistic Verification

```
procedure PROB-VERIFY($M$, lim)
    Entry[0] $\leftarrow$ initial states
    for $k \leftarrow [0 \ldots \text{lim}]$ do                $\triangleright$ iteration $[0 \ldots \text{lim}]$
        for all $s \in$ Entry[$k$] do
            if $s \notin$ Class[$i$], $\forall i \leq k$ then
                Insert $s$ into Class[$k$]
                Mod-DFS-visit($M, s, k$)
            end if
        end for
    end for
end procedure
procedure MOD-DFS-VISIT($M, s, k$)
    Push $s$ onto DFS-stack
    if $A(s) = \phi$ or $s$ violates some properties then
        Report error
    end if
    for all $(s, \alpha, t) \in trans(s)$ do
        if $\pi(s, \alpha, t) > 0$ then
            Insert $t$ into Entry[$k + \pi(s, \alpha, t)$]
        else
            if $t \in$ DFS-stack then
                Report livelock
            else if $t \notin$ Class[$i$], $\forall i \leq k$ and $t \notin I$ then
                Insert $t$ into Class[$k$]
                Mod-DFS-visit($M, t, k$)
            end if
        end if
    end for
    Pop DFS-stack
end procedure
```

in equivalence class $\mathcal{C}_0$, then the states in $\mathcal{C}_1$, and etc. It terminates when all the states in the transition system are explored, or when it uses up all the memory on the computer. We prove that after the algorithm finishes its k-th iteration, all states in $\mathcal{C}_k$ have been examined during that iteration.

Probabilistic verification consists of two procedures described in Algorithm 2. Prob-verify is the main procedure, which takes an MDP M, and a parameter *lim*. It contains a loop (line 3 - 10). We use iteration 0 to refer to the first pass of the loop, and iteration 1 refers to the second pass of the loop, etc. The subroutine Mod-DFS-visit implements the modified DFS. It takes the MDP M and two parameters s and k. The former is the starting point of DFS traversal, and the latter refers to the iteration that the procedure is currently in. The subroutine recursively traverse all the states that are reachable from s without traversing any low probability transition.

Class[.] and Entry[.] are two data structures that serve as containers for reachable states. Class[i] is used to store the states that have been traversed in iteration i, and Entry[i] stores the states from which we start the modified DFS in iteration i. Both containers can be implemented efficiently using hash table and state fingerprinting [14].

At the beginning of iteration 0, Entry[0] is initialized to the set of initial states. The algorithm uses Mod-DFS-visit to examine all the states that are reachable from every state $s \in$ Entry[0] without traversing any low probability transition. These states are being placed in Class[0]. All the states that are discovered by traversing a low probability transition are placed in Entry[j] for some $j > 0$, depending on the level of low probability transition.

In iteration 1, the algorithm starts the modified DFS from the states in Entry[1]. Again, all the states that are reachable without traversing any low probability transition are placed in the container Class[1], while those discovered after a low probability transition are placed in Entry[j] for some $j > 1$.

Next we take a closer look at Mod-DFS-visit. On entering the subroutine, the state s is push to a stack (line 13). When leaving the subroutine, s is popped from the stack (line 29). There are checks for deadlock and error before the transitions from s are examined (line 14). Note that we do not need to enumerate all the states in M before the algorithm starts. The set $trans(s)$ can be generated on-the-fly as the procedure proceeds by examining the state s.

The key difference between the modified DFS and the typical DFS is that whenever a state is reached by traversing a low probability transition, unlike typical DFS, we do not continue the recursive traversal on such state. Rather, the state is inserted into Entry[$k+\pi(.)$] (line 19), and will be traversed in later iteration of the algorithm (line 4). When Mod-DFS-visit subroutine terminates, it would have traversed all the states to which there exists at least one high probability path from the state where it was originally started.

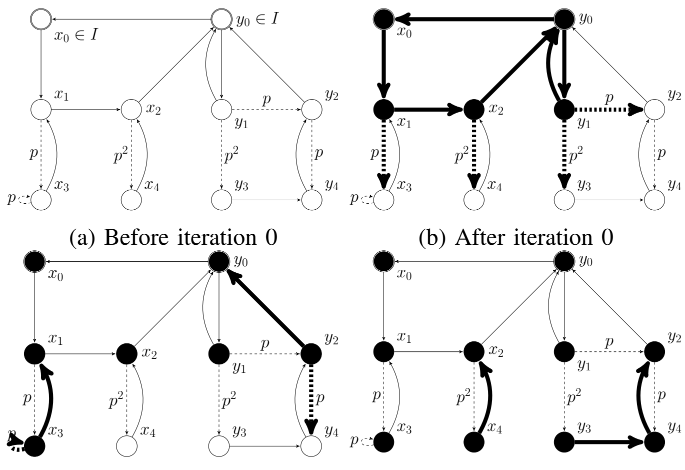
If a transition (s, α, t) is not a low probability transition, the modified DFS behaves the same as DFS. It first check if t already belongs to the DFS stack (line 22). If yes, then (s, α, t) is a backward edge, and the content in the stack constitutes a high probability cycle, which is a livelock according to our extended definition; otherwise, we then check if t is already stored in Class[m] for some $0 \leq m \leq k$ (line 24), if t is not explored yet, then t is placed into Class[k] and Mod-DFS-visit is called recursively to continue exploring the successors of t (line 25-26).

If, during the procedure, a deadlock or an error state is encountered, or an cycle is found, the algorithm halts immediately. To obtain an execution sequence that leads to error or deadlock as a counterexample, we could start typical DFS again from the initial state and traverse the states that have been stored in Class[.]. On reaching the error state, the counterexample is the content of the recursion stack.

A small example is shown in Fig.1.

- Iteration 0: Mod-DFS-visit starts from the initial states. x_0, x_1, x_2, y_0, y_1 are put into Class[0], x_3, y_2 are put into Entry[1], and x_4, y_3 are put into Entry[2].
- Iteration 1: Mod-DFS-visit starts from states in Entry[1]. x_3, y_2 are put into Class[1], and y_4 is put into Entry[2].
- Iteration 2: Mod-DFS-visit starts from states in Entry[2]. x_4, y_3, y_4 are put into Class[2], while Entry[i]= ϕ, $i > 2$.

The iterations are repeated until one of the following terminating condition occurs:



(a) Before iteration 0 (b) After iteration 0

(c) After iteration 1 (d) After iteration 2

Fig. 1: An example of verification

1) On finishing iteration k, there are not any states in Entry[i] for all $i > k$, in which case the state space of the transition system is fully searched.
2) The computer on which the algorithm executes is running out of memory to store the states in containers Class[.] and Entry[.].
3) When the algorithm finishes iteration *lim*, where *lim* is a parameter to the algorithm that specified the upper bound of equivalence class to be examined.

C. Proof of Correctness

When the verification algorithm terminates, we would want to make sure that all the explored states in Class[.] satisfy the properties to be checked. The following theorem and its corollaries assert that when iteration k terminates, it follows that all execution paths that contain states from classes $\mathcal{C}_0, \ldots, \mathcal{C}_k$ will not violate any property or include a high probability cycle. As a result, the first terminating condition implies that we have conducted a complete search on all the reachable states, and even though the second and the third terminating conditions leave the transition system to be partially explored, all the explored states are error-free.

Theorem 3: When iteration k terminates without reporting error, Class[k] = $\mathcal{C}_k$, $\forall k \geq 0$

This theorem implies that when iteration k finishes, all states in $\mathcal{C}_k$ have been checked for errors and placed into container Class[k]. The algorithm starts by traversing the states that are more likely to occur, which are states in equivalence class $\mathcal{C}_0$. As the iteration repeats, the equivalence classes are explored one after another, with the probability of the occurrence of states become smaller.

We shall prove the theorem by arguing that all the states in $\mathcal{C}_k$ can be reached from at least one state in Entry[k] via a high probability path. Hence, all states in $\mathcal{C}_k$ will be reached by modified-DFS starting from some state in Entry[k] and be placed in Class[k] during iteration k.

Proof. We prove the theorem by mathematical induction. First start with the case $k = 0$. In iteration 0, the modified depth-first search starts from the initial states. Let

$$T_0 = \{q | \exists \text{ high prob. path } \psi \text{ s. t. } \psi_0 \in I, \psi_{-1} = q\} \quad (2)$$

Note that as there exists at least one high probability path from s_0 to q, the modified DFS is guaranteed to encounter q when started from one of the initial states. When iteration 0 terminates, all states in T_0 will be placed into Class[0]. Recall the definition of equivalence class, $T_0 = \mathcal{C}_0$ and hence Class[0] = $\mathcal{C}_0$.

For the induction hypothesis, suppose that Class[k]= $\mathcal{C}_k$, $\forall k \leq m$. At the beginning of iteration $m+1$, line 19 of Algorithm 2 implies that

$$\begin{aligned} &\text{Entry}[m+1] \\ &= \{s' | \exists (s, \alpha, s') \text{ s. t. } s \in \text{Class}[n], \pi(s, \alpha, s') = m+1-n\} \end{aligned} \quad (3)$$

Note that not all states in Entry[$m+1$] belong to $\mathcal{C}_{m+1}$. The check on line 12 rules out those have already been reached earlier and were placed in Class[i] for some $i \leq m$. Let Entry[$m+1$]* be the set of states that survive the check, and Entry[$m+1$]$^* \subseteq \mathcal{C}_{m+1}$, as they can only be reached via an execution sequence σ such that $L(\sigma) \geq m+1$. For all $s \in$ Entry[$m+1$]*, there exists at least one execution sequence σ such that $L(\sigma) = m+1$ and $\sigma_{-1} = s$. Modified DFS only starts from the states in Entry[$m+1$]*.

By definition, a state t in $\mathcal{C}_{m+1}$ is reachable from one of the initial states through an execution sequence σ such that $L(\sigma) = m+1$, while there are no path such that $L(\sigma) < m+1$. Let $\sigma = s_0\alpha_0 s_1\alpha_1 \ldots \alpha_{n-1}s_n\alpha_n \ldots s_{n+j-1}\alpha_{n+j-1}s_{n+j}$, where $s_{n+j} = t$, $(s_{n-1}, \alpha_{n-1}, s_n)$ is a low probability transition, and $(s_n, \alpha_n, s_{n+1}), \ldots, (s_{n+j-1}, \alpha_{n+j-1}, s_{n+j})$ are high probability transitions. We have a length-j suffix $\psi = s_n\alpha_n \ldots s_{n+j-1}\alpha_{n+j-1}s_{n+j}$ being a high probability path, and s_n must belong to Entry[$m+1$]*.

The existence of high probability path ψ implies that for all states t' in $\mathcal{C}_{m+1}$, there exists a high probability path to t' from some state(s) in Entry[$m+1$]*. Modified DFS can reached t' recursively in iteration $m+1$ without putting t' into yet another container Entry[m'], $m' > m+1$. Therefore, every state t' in $\mathcal{C}_{m+1}$ and only the states in $\mathcal{C}_{m+1}$ will be examined and be placed into container Class[$m+1$] during iteration $m+1$. Therefore, when iteration $m+1$ terminates without being interrupted by discovering a state that represents system error, Class[$m+1$] = $\mathcal{C}_{m+1}$. □

Since the algorithm fills the container Class[0], Class[1], up to Class[k] in that order, we obtain the following corollaries immediately

Corollary 4: On finishing the k-th iteration, all states in $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k$ have been traversed and are checked for errors.

At this point, we know that all states in $\bigcup_{0 \leq i \leq k} \mathcal{C}_i$ satisfy all the property we want to check and are deadlock-free. We still need to know that the they are free of livelock.

Corollary 5: If the k-th iteration has terminated without reporting high probability cycle, then there is no high probability cycle among states in $\bigcup_{0 \leq i \leq k} \mathcal{C}_i$

Proof. First we know that within each equivalence class there is no high probability cycle, hence a cycle must contain states from different equivalence classes. Let the cycle be $\psi = s_1 s_2 \ldots s_n s_1$ and let s_i and s_j be the states on the cycle that are from different equivalence classes, specifically s_i in $\mathcal{C}_i$ and s_j in $\mathcal{C}_j$. Assume without losing generality that $j > i$. Then the path from s_i to s_j must contain at least one low probability transition. Otherwise, if the path is high probability path, s_i and s_j cannot be of different equivalence classes. Therefore, there cannot be any high probability cycle among states in the union of $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k$. □

Corollary 6: On finishing the k-th iteration, if Entry[i] is empty for all $i > k$, then the union of $\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_k$ is the complete transition system.

Finally, the last corollary states that if the algorithm terminates on condition 1, then the whole system is free of error.

Proof. Since Entry[i] is empty for all $i > k$, Class[i] is empty for all $i > k$, and hence $\mathcal{C}_i$ is empty for all $i > k$. □

IV. Probability bound computation

While we cannot determine the correctness of a system if not all the reachable states have been explored, it is possible to upper-bound the probability for the system to reach some error state. When verifying a transition system, we are interested in whether the system, when it starts from one of the initial states, will 1) return to any of the initial state or 2) arrives at any state that has not been explored by the verification algorithm, which can possibly lead to an error. We compute the upper-bound of the probability for the system to take some execution paths that end up as in the latter case. Originally, the bound is simply the product of the number of distinct execution sequences having the same number of low probability transitions, and the probability of taking such execution sequence [10]. By combining those execution sequences induced by probabilistic choices that have probabilities sum to one, the bound can be improved significantly by solving a linear program, whose constraints are obtained by examining the transitions of each explored state. We look at the linear program and justify that its optimal solution is indeed the desired probability bound.

Our objective is to bound the probability for the system to end up in the set of unexplored state when starting from the initial states. Suppose when the verification algorithm terminates, all states in class[0], …, class[k], or equivalently, states in $\mathcal{C}_0, \ldots, \mathcal{C}_k$, have been examined, while the states in class[$k+1$], class[$k+2$], …, have not been examined. The reachable states of the transition system can be divided into two disjoint sets: the set of explored states $T = \bigcup_{0 \leq i \leq k} \mathcal{C}_i$, and the set of unexplored states $U = \bigcup_{i > k} \mathcal{C}_i$.

Let $P_U(s)$ be the probability for the system to start the execution from state s and arrive at some state in U, instead of returning to any of the initial states. Then the probability bound for a system ending up in error

$$P_e \leq \max\{P_U(s_0) | s_0 \in I\} \quad (4)$$

In the following theorem, we shall see that the solution to a linear program yields an upper-bound to P_e.

Theorem 7: Let $((x_s^*)_{s \in T}, (y_{s|\alpha}^*)_{s \in T, \alpha \in A(s)}, z^*)$ be a solution to the following linear program. The solution to yields upper-bounds for P_e and $P_U(s), \forall s \in T$. Specifically, $z^* \geq P_e$ and $x^*_s \geq P_U(s), \forall s \in T$.

$$\begin{aligned}
&\mathbf{min}\ z\\
&\mathbf{s.\ t.}\ z \geq x_{s_0},\ \forall s_0 \in I\\
&\qquad 0 \leq x_s \leq 1,\ \forall s \in T\\
&\qquad x_s \geq \sum_{t \in \alpha\text{-succ}(s) \cup T} q_{(s,\alpha,t)} x_t + \sum_{u \in \alpha\text{-succ}(s) \cup U} q_{(s,\alpha,u)}\\
&\qquad\qquad + y_{s|\alpha},\ \forall s \in T, \alpha \in A(s)\\
&\qquad y_{s|\alpha} \geq x_b,\ \forall\ b \in \alpha\text{-succ}(s), \forall s \in T, \alpha \in A(s)
\end{aligned} \tag{5}$$

where

$$q_{(v,\alpha,w)} = \begin{cases} 0 & \text{if } \pi(v,\alpha,w) = 0 \vee w \in I \\ \hat{p}^{\pi(v,\alpha,w)} & \text{if } \pi(v,\alpha,w) > 0 \end{cases} \tag{6}$$

Proof. Assuming that $x^*_s \geq P_U(s), \forall s \in T$. The first set of constraints requires that $z^* \geq x^*_{s_0} \geq P_U(s_0), \forall\ s_0 \in I$, and thus $z^* \geq \max_{s_0 \in I}\{P_U(s_0)\} \geq P_e$.

The second set of constraints ensures that the value of $(x_s)^*_{s \in T}$ in a solution is a probability.

The third set of constraints and the fourth set of constraints should be considered simultaneously. They account for the probability relation between transitions for a given state.

First consider $P_U(s)$ in an MDP with explicit probability distribution for all probabilistic choices. The semantic of MDP implies that the transition from some state s can be separated into selecting a non-deterministic choice and selecting a probabilistic choice according to probability distribution. Therefore

$$P_U(s) = \sum_{\alpha \in A(s)} \mathbb{P}\{\alpha \text{ is selected}\} P_U(s|\alpha) \tag{7}$$

In most probabilistic model checking techniques [13], non-deterministic choices are resolved using adversaries (or equivalently, schedulers). For any adversary resolution, the selection among non-deterministic choices should satisfy $\sum_{\alpha \in A(s)} \mathbb{P}\{\alpha \text{ is selected}\} = 1$. Then (7) becomes

$$P_U(s) \leq \max_{\alpha \in A(s)} \{P_U(s|\alpha)\} \tag{8}$$

Suppose that the transitions have explicit probability distribution. Given that a non-deterministic action $\alpha \in A(s)$ is selected, we have

$$P_U(s|\alpha) = p_{(s,\alpha,t_0)} P_U(t_0) + \ldots + p_{(s,\alpha,t_n)} P_U(t_n) \tag{9}$$

, where $t_0, \ldots, t_n$ are the α-successors of s with initial states removed. The paths toward initial states do not contribute to the probability of reaching unexplored state. If t is an unexplored state, that is, $t \in U$, then $P_U(t) = 1$.

Since the skewed-probability MDPs do not come with explicit probability distribution for all probabilistic choices, (9) does not apply anymore. Fortunately, we can still find the largest possible values of $P_U(s|\alpha)$ and $P_U(s)$. Consider possible transitions from state s after an action α is selected in Fig.2. The transitions to the initial states are eliminated as they do not contribute to the probability of reaching an unexplored state. Given that α is selected, state s has n high probability transitions that lead to state $b_1, \ldots, b_n$, m low probability transitions with various levels leading to state $c_1, \ldots, c_m$, and r low probability transitions with various levels leading to some unexplored states not shown in the figure. Note that $b_1, \ldots, b_n, c_1, \ldots, c_m$ are distinct. By definition, we have $p_{(v,\alpha,w)} \leq \hat{p}^{\pi(v,\alpha,w)}$, and $\sum_{w \in \alpha\text{-succ}(v)} p_{(v,\alpha,w)} = 1$, so we can upper-bound $P_U(s|\alpha)$:

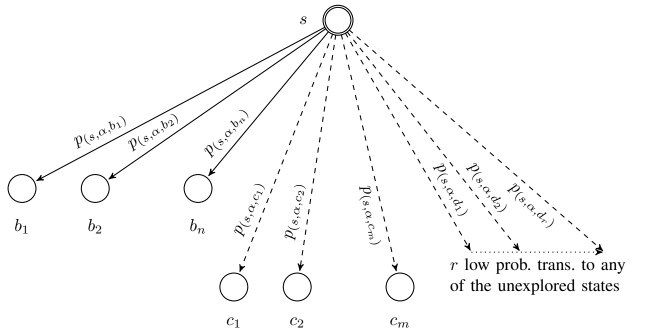


Fig. 2: Possible transitions for a given state s

$$\begin{aligned}
&P_U(s|\alpha)\\
&= p_{(s,\alpha,b_1)} P_U(b_1) + \ldots + p_{(s,\alpha,b_n)} P_U(b_n) + p_{(s,\alpha,c_1)} P_U(c_1)\\
&\quad + \ldots + p_{(s,\alpha,c_m)} P_U(c_m) + p_{(s,\alpha,d_1)} + \ldots + p_{(s,\alpha,d_r)}\\
&\leq \max\{P_U(b_1), \ldots, P_U(b_n)\} + q_{(s,\alpha,c_1)} P_U(c_1)\\
&\quad + \ldots + q_{(s,\alpha,c_m)} P_U(c_m) + q_{(s,\alpha,d_1)} + \ldots + q_{(s,\alpha,d_r)}
\end{aligned} \tag{10}$$

Substituting (10) into (8) yields the set of inequalities for each state $s \in T$

$$P_U(s) \leq \max_{\alpha \in A(s)} \left[\begin{array}{l} \max\{P_U(b_1), \ldots, P_U(b_n)\} \\ +q_{(s,\alpha,c_1)} P_U(c_1) + \ldots + q_{(s,\alpha,c_m)} P_U(c_m) \\ +q_{(s,\alpha,d_1)} + \ldots + q_{(s,\alpha,d_r)} \end{array} \right] \tag{11}$$

Fixing state s and a selected non-deterministic action α, the fourth set of constraints require that $y^*_{s|\alpha} \geq x^*_b \geq P_U(b)$ for all state b being the α-successors of s. Therefore in the solution, $y^*_{s|\alpha} \geq \max\{P_U(b_1), \ldots, P_U(b_n)\}$ is satisfied. Then (11) implies that, for all $\alpha \in A(s)$,

$$x^*_s \geq \left[\begin{array}{l} y^*_{s|\alpha} \\ +q_{(s,\alpha,c_1)} x^*_{c_1} + \ldots + q_{(s,\alpha,c_m)} x^*_{c_m} \\ +q_{(s,\alpha,d_1)} + \ldots + q_{(s,\alpha,d_r)} \end{array} \right] \geq P_U(s) \tag{12}$$

We now have the third set of constraints. $\square$.

Finally, we can use linear program solver to find the optimal solution, in which the minimum value of the variable z^* will be the tightest bound for P_e. Moreover, the set of constraints are reasonably sparse, which gives ways for the solver to employ more sophisticated algorithms.

A probability bound obtained by solving a linear optimization problem yields a bound that is as tight as possible without violating the constraints reflected by the information we have on probabilistic choices. In the predecessor of probabilistic verification, a bound is obtained by counting the number of distinct paths originated at the initial states and end up in the unexplored state. This number can potentially be exponential in the number of explored states.

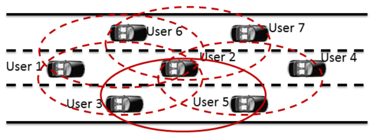


Fig. 3: The 7-party lock protocol

V. Experiment: The lock protocol

We apply probabilistic verification to the lock protocol [15], which is used as a subroutine in an collaborative vehicle merge protocol. The 7-party lock protocol has a large state space that prevents most model checker from complete verification. We demonstrate that probabilistic verification is able verify the lock protocol and obtain a bound on error probability, while the PRISM model checker [13] cannot complete the verification. The probability bound it computes is significantly tighter than that produced by its predecessor.

The collaborative vehicle merge protocol used the lock protocol to create non-overlapping groups among a set of users, as shown in Fig.3. Specifically, user 2, user 3, user 5, and user 6 are attempting to form a group with two neighboring users, shown as dashed circle. Up to one group can be formed without overlapping with other groups, represented by the solid circle. The implementation of the lock protocol on each vehicle interacts with each other through an unreliable communication channel that is prone to message loss. The probabilistic choices associated with message loss are designated as level-1 low probability choice with probability $< \hat{p} = 10^{-4}$.

We verify 1) if the lock protocol satisfies mutual exclusion property, i.e., a user cannot participate in two or more groups at the same time, and 2) that the requesting user is only notified of the success of group creation only after the other group members have agreed to participate in the group.

PRISM is a well-known model checker in terms of probabilistic model checking. It is able to determine the probability for a prespecified property to hold or not, given that it is able to generate all the reachable states. To ensure that the comparison is meaningful, no optimization option in PRISM is enabled, as there are not any optimization techniques employed in current implementation of probabilistic verification. Unfortunately, before PRISM can produce an diagnosis of the system, it aborts by using up all the available memory.

Next, we apply probabilistic verification to the system. Although probabilistic verification is unable to completely examine all the reachable states and to determine if either of the properties is violated, it explores 343,207 states in equivalent classes $\mathcal{C}_0, \ldots, \mathcal{C}_3$, and 4,522,358 state transitions. Instead of aborting, it computes a upper-bound for probability of violating either of the properties to be 2.97×10^{-14}. Instead, the predecessor algorithm produced a bound of 4×10^{-9}. The linear program improves the bound by a factor of more than 10^5.

VI. Conclusion

Probabilistic verification provides an alternative to mitigate the state space explosion problem. It does not require that all reachable states be examined, and a subset of the states would be sufficient. It produces a bound for the probability of reaching those states that have not been traversed. It is able to verify systems with a large number of reachable states, such as the lock protocol, while model checkers that require all the reachable states to be traversed cannot. Furthermore, it is a suitable tool for systems that interact with the physical world, from which many failure scenarios may arise.

However, our goal is not to replace the existing model checkers, but to complement them. Probabilistic verification techniques can be used in conjunction with the existing optimization methods that also aim at tackling large state space. Our next step is to explore the possibility of integrating probabilistic verification into the well-used model checkers such as PRISM and SPIN [16], and eventually increase the range of systems to which these model checkers are applicable.

Acknowledgment

This work is supported by NSF grants CNS-1329593.

References

[1] A. P. Sistla and P. Godefroid, "Symmetry and reduced symmetry in model checking," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 4, pp. 702–734, 2004.

[2] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper, "Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem," 1996.

[3] S. Katz and D. Peled, "Verification of distributed programs using representative interleaving sequences," *Distributed Computing*, vol. 6, no. 2, pp. 107–120, 1992.

[4] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer Berlin Heidelberg, 2000, pp. 154–169.

[5] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 58–70, 2002.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang, "Symbolic model checking: 10 20 states and beyond," in *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*. IEEE, 1990, pp. 428–439.

[7] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001.

[8] H. Günther and G. Weissenbacher, "Incremental bounded software model checking," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. ACM, 2014, pp. 40–47.

[9] A. Udupa, A. Desai, and S. Rajamani, "Depth bounded explicit-state model checking," in *Model Checking Software*. Springer, 2011, pp. 57–74.

[10] N. F. Maxemchuk and K. Sabnani, "Probabilistic verification of communication protocols," *Distributed Computing*, vol. 3, no. 3, pp. 118–129, 1989.

[11] S.-p. Lin, Y. Gu, and N. F. Maxemchuk, "A multiple stack architecture for intelligent vehicles," in *Intelligent Vehicles Symposium Proceedings, 2014 IEEE*. IEEE, 2014, pp. 268–273.

[12] C. Baier and J.-P. Katoen, "Principles of model checking," 2008.

[13] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker, "Automated verification techniques for probabilistic systems," in *Formal Methods for Eternal Networked Software Systems*. Springer, 2011, pp. 53–113.

[14] G. J. Holzmann, "An analysis of bitstate hashing," *Formal methods in system design*, vol. 13, no. 3, pp. 289–307, 1998.

[15] S.-p. Lin and N. F. Maxemchuk, "The fail-safe operation of collaborative driving systems," *Journal of Intelligent Transportation Systems*, no. ahead-of-print, pp. 1–14, 2014.

[16] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.