Efficient State Classification of Finite State Markov Chains †

Aiguo Xie and Peter A. Beerel
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562

Abstract This paper presents an efficient method for state classification of finite state Markov chains using BDD-based symbolic techniques. The method exploits the fundamental properties of a Markov chain and classifies the state space by iteratively applying reachability analysis. We compare our method with the current state-of-the-art technique which requires the computation of the transitive closure of the transition relation of a Markov chain. Experiments in over a dozen synchronous and asynchronous systems demonstrate that our method dramatically reduces the CPU time needed, and solves much larger problems because of reduced memory requirements.

1 Introduction

Markov chains are an important class of stochastic processes that can effectively model randomly evolving systems. They are conceptually simple because they have *short memory*, i.e., their future evolution depends only on the current outcome and not on further history. Although they are simple, Markov chains have found vast applications in engineering including the analysis of system reliability [1, 2], control, e.g., [3], power consumption [4, 5] and performance [6, 7, 8, 9, 10]. The first two applications are usually concerned with the short-term (*transient*) behavior or *time to certain events* of the Markov chain whereas the latter two applications often involve knowledge of the long-term (*stationary*) behavior. An important and typically the first step in carrying out these analyses is the structural analysis of the Markov chain. It partitions the state space into two sets, namely, the set of *transient states* (those expected to be visited only a finite number of times) and the set of *recurrent states* (those expected to be re-visited infinitely often once they are visited). The set of recurrent states may be further classified into different classes which once entered trap the Markov chain inside the class indefinitely. Both transient and stationary analyses need these structural information in order to proceed or get simplified. The purpose of this paper is to present an efficient method that performs this state classification step.

†This work is funded in part by a NSF CAREER Award MIP-9502386 and a gift from the Intel Corporation.

The structure of the state space of a Markov chain may be represented by a directed graph where vertices correspond to the states and edges correspond to the transitions among the states. One method to perform structural analysis of the state space is to *explicitly* identify all the (maximal) *strongly connected components* (SCCs) of the graph. Components (those called terminal SCCs) that do not have edges leading to other components correspond to the recurrent state classes and all other components correspond to the set of transient states. Since Markov chains derived from real systems often possess huge state space [6, 11, 8], this method can be intolerably slow because it analyzes states *sequentially*. Another method for structural analysis is described in [11][1] which uses *implicit* (symbolic) techniques based on *binary decision diagrams* (BDDs) [12] so that all the SCCs of the corresponding graph are *concurrently* computed and the terminal SCCs are subsequently identified. A key step of this method uses the recursive paradigm in [13] to compute all the states that can be reached from individual states, or from a graph point of view, the *transitive closure* of the corresponding graph. Unfortunately, this step tends to be *memory-intensive* and *time-consuming*.

This paper also proposes a BDD-based symbolic technique but exploits the fundamental property of transience and recurrence of a Markov chain. Our technique *sequentially* explores the state space by repeatedly applying symbolic reachability analysis. Newly reached transient states and recurrence class are detected after each reachability analysis. Since most systems contain only a few recurrence classes and a large portion of their state spaces are recurrent, the number of times reachability analysis is performed is often very small. Finally, because reachability analysis is typically much faster and less memory-intensive than transitive closure computation, our method dramatically outperforms the one discussed in [11] on almost all the examples we have tested. In particular, we observe run time reduction of up to several orders of magnitude, and our method solves all the examples on which the previous method

[1]We note, however, that the main thrust of [11] is symbolic algorithms for stationary analysis with less emphasis on structural analysis.

fails due to insufficient memory.

The remaining of this paper is organized as follows. Section 2 reviews basics of Markov chains and the previous work on their structural analysis. Section 3 describes the principles, proofs, and the algorithm of our method. Experimental results are given in Section 4 and our conclusions are given in Section 5.

2 Preliminaries and Previous Work

2.1 Finite-state Markov chains

Consider a discrete state space $S = \{1, 2, \cdots\}$. A discrete-time stochastic process X on S is a sequence of random variables, $\{X_n \in S : n \geq 0\}$. We call the value assumed by random variable X_n the state of X at time instant n. X is a *Markov chain* (or *Markovian*) if its future evolution depends only on its current state. More formally, if X is Markovian and it is in state $i \in S$ at time n, then the probability that state $j \in S$ is visited at time $n+1$ is independent of its history before time n, i.e, $Pr(X_{n+1} = j \mid X_n = i, (X_{n-1}, \cdots, X_0) \in S) = Pr(X_{n+1} = j \mid X_n = i)$. X is *time-homogeneous* if the above conditional probability is independent of time n. In this case, the evolution of X is governed by a (1-step) *transition probability matrix* $P = (p_{ij}), i, j \in S$ where $p_{ij} = Pr(X_{n+1} = j \mid X_n = i) = Pr(X_1 = j \mid X_0 = i)$.

Real systems can often be modeled as Markov chains which are time-homogeneous. We assume the state space S is finite so that one may write $S = \{1, \cdots, N\}$ where N is the size of S. Let $M = (S, P)$ denote such a chain with probability transition matrix P. Systems with massive concurrency [6] or having correlations among system events [4] can all be modeled by Markov chains with properly defined state spaces.

To consider the relationship between states of S, one needs the notion of *communication*. A state $i \in S$ communicates with state $j \in S$, denoted by $i \rightarrow j$, if there is a positive probability that state j will eventually be visited once state i is visited. Or equivalently, $p_{ij}(n) = Pr(X_n = j \mid X_0 = i) > 0$ for some $n > 0$. Otherwise, state i does not communicate with state j, denoted by $i \not\rightarrow j$. States i and j *intercommunicate* if $i \rightarrow j$ and $j \rightarrow i$, written $i \leftrightarrow j$.

Definition 2.1 *State $i \in S$ is recurrent if $\forall j \in S$ such that $i \rightarrow j$, $j \rightarrow i$ as well. Otherwise, state i is transient, i.e, $\exists j \in S$ such that $i \rightarrow j$ but $j \not\rightarrow i$.*

It can be easily checked that a recurrent state is visited infinitely often once it is visited, while the expected number of the times M visits a transient state is bounded from above.

Many of the nice properties of Markov chain M can be derived from the famous *Chapman-Kolmogorov* theorem [14] which expresses the probability of M to go from one state to another in $n \geq 0$ time steps in terms of the 1-step transition matrix P. One such property concerning the communication relation is as follows.

Theorem 2.1 *If $i \rightarrow j$ and $j \rightarrow k$, then $i \rightarrow k$. Further, if $i \leftrightarrow j$, then $j \leftrightarrow i$.*

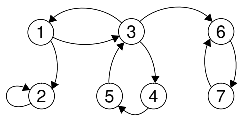


Figure 1: A small Markov chain example.

By Definition 2.1 and Theorem 2.1, it is easily verified that if states i and j intercommunicate, either both of them are recurrent or both of them are transient.

Theorem 2.1 ensures that $\leftrightarrow$ is an equivalence relation so that the state space can be partitioned into equivalence classes according to $\leftrightarrow$. That is, $S = T \cup R_1 \cup R_2 \cdots \cup R_K$ where T is the set of transient states, R_1 through R_K are K recurrence classes each containing recurrent states that do not communicate with states in other classes.

Example Figure 1 shows a Markov chain containing 7 states. State 1 is clearly transient because $1 \rightarrow 2$ but $2 \not\rightarrow 1$. States 3, 4 and 5 are also transient for similar reasons. The chain has two recurrence classes, namely, $\{2\}$ and $\{6, 7\}$. □

We have seen that the transience and recurrence property of states can be derived from the their communication relation. In particular, the structure of the state space can be represented by a directed graph $G = (V, E)$, where the vertices in V are in one-to-one correspondence with the states in S. There is an edge e_{uv} from vertex u to v iff $p_{ij} > 0$ where the vertices u and v correspond to the states i and j. Then, $i \rightarrow j, i, j \in S$ means there is a path from vertex u to v in G. Similarly, $i \leftrightarrow j$ means vertices u and v are strongly connected. We call graph G the boolean transition graph of M.

Many types of analysis may be done on the Markov chain M. These analyses include the computation of the probability distribution of M over its state space at a fixed time instant (*transient analysis*), the expected time and probability to enter a set of states (*first-passage-time and absorption probability analysis*) [15], and its asymptotic probability distribution after an infinite amount of time (*stationary analysis*). State classification plays plays an important role in these analyses. For instance, first-passage-time analysis is often performed against a particular recurrence class. By treating each recurrence class as a single state, absorption analysis may also be accelerated. Moreover, the stationary analysis is not possible without the identification of the recurrence classes (unless M is guaranteed to have only one recurrence class) because the system will be underdetermined if the chain has multiple recurrence classes. In fact, even when the number of the recurrence classes is expected to be one, it is often necessary to verify this is the case because modeling errors or design bugs may introduce extra recurrence classes.

2.2 Previous work

We focus on the recently proposed symbolic method [11] which finds the recurrent classes by *concurrently* identifying all terminal SCCs of the transition graph G

associated with the Markov chain.

The method symbolically computes *all* the SCCs in G by creating the transitive closure TC of G which describes the set of states that each state of S communicates with. One may think of TC as a boolean matrix of size $S \times S$, i.e, $TC_{ij} = \begin{cases} 1 & if \quad i \to j, \\ 0 & otherwise. \end{cases}$ Therefore, checking if $i \to j$ is equivalent to checking if the element $TC_{ij} = 1$. Suppose it is determined that $i \to j$ by the fact that $T_{ij} = 1$. To see if $j \to i$ so that i, j are strongly connected, one merely needs to check if $T_{ji} = 1$. Thus, the second step in the method is to *term-wise* multiply TC with its transpose, TC^t, yielding a symmetric matrix (let it be denoted by $TC{*}TC^t$). Clearly, i and j are in the same SCC *iff* $(TC * TC^t)_{ij} = (TC^t * TC)_{ji} = 1$. All above operations are performed symbolically.

Next, each *SCC* is represented by a particular state (called the representative state) of the SCC using the so called *datum priority function*[16] of the corresponding BDD. A new directed graph consisting of these representative states is then constructed so that it has an edge from one (representative) state i to another (representative) state j *iff* there is an edge from some state in the SCC represented by i to some state in the SCC represented by j. This new graph is *acyclic* and its leaf vertices represent the terminal SCCs.

The bottleneck of the method is the computation of the transitive closure TC of G. To this end, a recursive algorithm presented in [13] which also uses BDD-based symbolic operations is adopted. Nevertheless, it can still take intensive amount of CPU time and memory to compute TC when G is large.

3 State classification by iterative application of reachability analysis

The method we present in this section does not compute the transitive closure of the graph G. Rather, it relies on the definition of a transient/recurrent state. Once we determine a state to be transient/recurrent, the transience and recurrence properties of other states that communicate with this state are deduced and they are removed from further consideration.

3.1 The principles

We start by defining the *forward* and *backward* sets of a state.

Definition 3.1 *The forward set of state $i \in S$, denoted by $\mathcal{F}(i)$, is the set of states that i communicates with. That is, $\mathcal{F}(i) = \{j \in S \mid i \to j\}$. Similarly, the backward set of state i, denoted by $\mathcal{B}(i)$, is the set of states that communicate with i. That is, $\mathcal{B}(i) = \{j \in S \mid j \to i\}$.*

The following lemma shows a simple property of forward and backward sets.

Lemma 3.1 *Let $i, j \in S$. If $j \in \mathcal{F}(i)$, then $\mathcal{F}(j) \subseteq \mathcal{F}(i)$. Similarly, if $j \in \mathcal{B}(i)$, then $\mathcal{B}(j) \subseteq \mathcal{B}(i)$.*

Proof The proof is a direct consequence of the transitivity property of relation $\to$ given in Theorem 2.1.

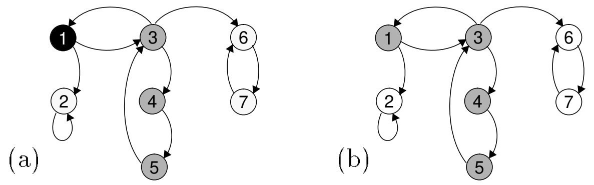


Figure 2: Determining if a state is recurrent: (a) computing $\mathcal{F}(1)$ and $\mathcal{B}(1)$, (b) $\{1\} \cup \mathcal{B}(1)$ all transient.

Suppose $j \in \mathcal{F}(i)$, i.e., $i \to j$, then, $\forall k \in \mathcal{F}(j)$, i.e., $j \to k$, we have $i \to k$, meaning $k \in \mathcal{F}(i)$ so that $\mathcal{F}(j) \subseteq \mathcal{F}(i)$. Similarly, if $j \in \mathcal{B}(i)$, i.e., $j \to i$, then, $\forall k \in \mathcal{B}(j)$, i.e., $k \to j$, we have $k \to i$, which implies $\mathcal{B}(j) \subseteq \mathcal{B}(i)$. $\Box$

Theorem 3.1 *A state $i \in S$ is recurrent if and only if $\mathcal{F}(i) \subseteq \mathcal{B}(i)$. In other words, i is transient if and only if $\mathcal{F}(i) \not\subseteq \mathcal{B}(i)$.*

Proof Suppose $i \in S$ is recurrent, then $\forall j \in S$ such that $i \to j$, i.e., $j \in \mathcal{F}(i)$, we have $j \to i$ by Definition 2.1 of a recurrent state. That is to say, $j \in \mathcal{B}(i)$. Therefore, $\mathcal{F}(i) \subseteq \mathcal{B}(i)$. Conversely, if $\mathcal{F}(i) \subseteq \mathcal{B}(i)$, which means $\forall j \in S$ such that $i \to j$, we have $j \in \mathcal{F}(i) \subseteq \mathcal{B}(i)$, and hence $j \to i$. Then i is recurrent by Definition 2.1. The fact that a state is either recurrent or transient concludes the proof. $\Box$

Theorem 3.2 *If state $i \in S$ is transient, then states in $\mathcal{B}(i)$ are all transient. If state i is recurrent, on the other hand, states in $\mathcal{F}(i)$ are all recurrent. In the latter case, set $\mathcal{F}(i)$ is a recurrence class, and set $\mathcal{B}(i) - \mathcal{F}(i)$ (if not empty) contains only transient states.*

Proof Suppose state i is transient. By Theorem 3.1, $\mathcal{F}(i) \not\subseteq \mathcal{B}(i)$, i.e., $\exists k \in \mathcal{F}(i)$ such that $k \notin \mathcal{B}(i)$. Now, suppose state $j \in \mathcal{B}(i)$, then $j \to k$ so that $k \in \mathcal{F}(j)$. This is because $i \in \mathcal{F}(j)$ so that $\mathcal{F}(i) \subseteq \mathcal{F}(j)$ by Lemma 3.1. On the other hand, $\mathcal{B}(j) \subseteq \mathcal{B}(i)$ since $j \in \mathcal{B}(i)$. Therefore, we have state $k \in \mathcal{F}(j)$ but $k \notin \mathcal{B}(j)$ since $k \notin \mathcal{B}(i)$, which implies $\mathcal{F}(j) \not\subseteq \mathcal{B}(j)$ so that j is transient by Theorem 3.1.

If state i is recurrent, i.e., $\mathcal{F}(i) \subseteq \mathcal{B}(i)$ by Theorem 3.1, then, $\forall j \in \mathcal{F}(i) \Longrightarrow i \leftrightarrow j$ by definition so that we have $\mathcal{F}(j) \subseteq \mathcal{F}(i)$ and $\mathcal{B}(i) \subseteq \mathcal{B}(j)$ by Lemma 3.1. Thus, $\mathcal{F}(j) \subseteq \mathcal{F}(i) \subseteq \mathcal{B}(i) \subseteq \mathcal{B}(j)$, which implies j is recurrent by Theorem 3.1. Finally, if i is recurrent and $\mathcal{B}(i) - \mathcal{F}(i)$ is not empty, let state $k \in \mathcal{B}(i) - \mathcal{F}(i)$, we merely need to show that $\mathcal{F}(k) \not\subseteq \mathcal{B}(k)$ so that k is transient. In fact, $k \in \mathcal{B}(i) \Longleftrightarrow i \in \mathcal{F}(k)$, and $k \notin \mathcal{F}(i) \Longleftrightarrow i \notin \mathcal{B}(k)$, which implies $\mathcal{F}(k) \not\subseteq \mathcal{B}(k)$. $\Box$.

Example Figure 2(a) shows the small example Markov chain in Section 2. To determine if state 1 is recurrent, we first compute $\mathcal{F}(1) = \{1, 2, 3, 4, 5, 6, 7\}$ and $\mathcal{B}(1) = \{1, 3, 4, 5\}$. Since $\mathcal{F}(1) \not\supseteq \mathcal{B}(1)$, not only is state 1 transient, all states in $\mathcal{B}(1)$ are transient as well. Thus, only states 2, 6 and 7 need to be further classified. $\Box$

Theorem 3.1 determines if a state i is recurrent by checking if its forward set is contained in its backward set. If it is, then a recurrence class has been found which equals the forward set. Moreover, according to Theorem

```
forward_set(s, Q) {
    RS^(0) ← ∅, FS^(0) ← {s};
    k ← 0;
    while(FS^(k) ≠ ∅) {
        FS^(k+1)(x) = ∃_y FS^(k)(y) Q(y,x) ∧ ¬RS^(k)(x);
        RS^(k+1)(x) = RS^(k)(x) ∨ FS^(k+1)(x);
        k ← k + 1; }
    return(F(s) ← RS^(k)); }

backward_set(s, Q) {
    RS^(0) ← ∅, FS^(0) ← {s};
    k ← 0;
    while(FS^(k) ≠ ∅) {
        FS^(k+1)(x) = ∃_y FS^(k)(y) Q(x,y) ∧ ¬RS^(k)(x);
        RS^(k+1)(x) = RS^(k)(x) ∨ FS^(k+1)(x);
        k ← k + 1; }
    return(B(s) ← RS^(k)); }
```

Figure 3: Computing the forward set and backward set.

3.2 if the backward set properly contains the forward set, those states in the backward set not belonging to the forward set are all found to be transient. In the case the forward set is not contained in the backward set, we have found a set of transient states equal to $\{i\} \cup \mathcal{B}(i)$.

The next subsection gives our symbolic algorithm for state classification based on the above theory.

3.2 The algorithm

We associate the transition graph $G = (S, E)$ with a boolean transition relation Q defined from $S \times S$ into $\{0, 1\}$ such that $Q(s, s') = 1$ iff $e_{s,s'} \in E$. We say s is the current state while s' is the next state. To represent the relation Q with BDDs, let X and Y be two sets of boolean variables encoding current states and next states, respectively. Details of BDDs and their operations can be found in [12].

3.2.1 Computing forward and backward sets

Computation of the forward set can be performed exactly as the symbolic reachability analysis for finite state machines using *fixed-point calculations* [17, 18].

The idea of fixed point calculation may be explained by the concept of the frontier set, FS. Initially, the reachable state set, RS, is set to $\emptyset$, and the frontier set is set to be the initial states. That is, $RS^{(0)} = \emptyset$, and $FS^{(0)} = I$, where I is the set of initial states. Next, the set of states, Z, that can be reached from the current frontier set $FS^{(0)}$ in one time step (or one state transition) is computed. Those states that are newly reached during this iteration are taken as the frontier for the next iteration. The process repeats until the frontier set is empty.

The first procedure given in Figure 3 computes the forward set of state s where $\exists$ and $\wedge$ denote BDD operations of existential quantification and AND, respectively. Applying the same idea, the backward set of s can be computed similarly.

3.2.2 Classifying states

Figure 4 shows our state classification algorithm. It *iteratively* finds all the transient states and the recurrence classes. During each iteration, a state from the remaining state space S' is taken as a *trial* state whose forward set and backward set are computed using the procedures given in Figure 3. Next, the trial state is determined to be either recurrent or transient by checking the containment of its forward and backward sets as discussed in the Section 3.1. States that communicate with this trial state are also determined for their recurrence/transience property. The iteration terminates when there is no state to be classified.

```
State_classification(S, Q) {
    S' ← S;
    while(S' ≠ ∅) {
        s ← random_take(S');
        F(s) ← forward_set(s, Q);
        B(s) ← backward_set(s, Q);
        if(F(s) ∧ ¬B(s) = ∅) {
            report  F(s)  a recurrence class;
            report  B(s) ∧ ¬F(s)  all transient; }
        else
            report  s ∨ B(s)  all transient;
        S' ← S' ∧ ¬(s ∨ B(s)); } }
```

Figure 4: The state classification algorithm.

The function $random\_take(S')$ works as follows. It randomly takes a state, say s_1 from S' as a start point, and searches for a state s_2 not yet considered to be a trial state but reachable from s_1 within a user definable number of steps. The intuition is that, state s_2 is more likely to be recurrent because $s_2 \in \mathcal{F}(s_1)$. Since this search can be done iteratively and after each iteration only one state is selected to be the new frontier set, the time spent in this function is negligible.

In general, our algorithm has much better complexity than the method in [11]. This is because the reachability analysis is performed only for part of the state space, i.e, for those trial states returned by function *random_take*() whereas the transitive closure of graph Q required by the latter is equivalent to the knowledge of the reachable sets for all the states in S.

The worst case of our algorithm may occur if each iteration determines the transience or recurrence property of only the corresponding trial state when its backward set contains at most itself. Then, exactly $|S|$ iterations would be required. This could happen for instance if (1) every state in S forms a single-state recurrence class or (2) all states except one are transient and each transient state reaches only the sole recurrent state. Since most real systems contains relatively few recurrence classes and a significant portion of the state space is recurrent, very few (typically only one) iterations are performed so that the worst case complexity is rarely observed. Another case where our algorithm may also be slow is when all states (nearly) form a loop. This case does not require a large number of iterations, but the time spent in the reachability analysis during each iteration could be significant. However, the time complexity of our algorithm still remains at the same order of that of the reachability analysis. The *counter* example in the next section is a case of this kind.

4 Experimental Results

We implemented both our method and the method in [11] based on the CUDD [19] and SMV [20] packages on a SPARC20 with 256 Megabytes of memory. The efficiencies of the two methods are compared on many circuits as well as a queueing network model. All of them are specified in SMV format.

4.1 Synchronous and asynchronous circuits

Our synchronous examples are the synchronous bus arbiter (*syncarb*) [20], the mutual exclusion element (*mutex*), the counter circuit (*counter*), and a 3-stage synchronized pipeline (*periodic*) which are all from the SMV package. Our asynchronous examples include the distributed mutual exclusion (*dme*) circuit [20], the pausible clock interface (*pci*) [21], the asynchronous differential equation solver [21] with its estimated version (*diffeq_est*) and its back-annotated version (*diffeq_bka*) both of which were analyzed symbolically for their average performance in [8] and finally the asynchronous FIFO circuit *fifo* also discussed in [8].

According to the results in Table 1, all the circuits contain a single recurrence class. The synchronous ones have no transient states. Among the asynchronous ones, only the *diffeq* and the *fifo* contain a significant fraction of transient states.

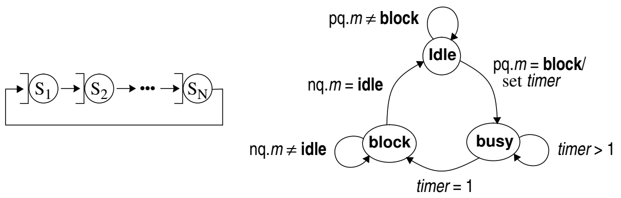
Our method finishes all examples while the previous method fails on half of them due to insufficient memory. Comparing the run time, except on the counter circuit, our method dramatically outperforms the previous method. With more experiments, we notice that for the *fifo* example, the CPU time required by our method increases polynomially while the state space grows exponentially. This is because the BDD size of its transition relation Q grows polynomially with each added stage. Similar performance is achieved in the *syncarb*, *pci* and *dme* examples. The exception of the counters is because their state spaces are complete graphs for which the hashed recursive TC computation is particularly efficient [13].

4.2 Closed queueing networks

Finally, we consider a behavioral model of a *closed queueing network* [22]. Each queue is modeled as a single server with finite storage capacity. For simplicity, we assume each server has no input/output buffers. We model the delay of the server with a bounded time interval with a discrete probability distribution. In practice, this delay model is more realistic than the typical models with exponential delays. Figure 5(a) shows the overall system structure and Figure 5(b) specifies the behavior of a single server as a finite state machine.

Notice that because of its closed nature of the system, once (randomly) initialized, the number of jobs in the system remains constant. Consequently, this example has multiple recurrence classes.

Initially, a server may or may not have a job to process, i.e, it is randomly initialized as either `idle` or `busy`. The server leaves `idle` mode if there is an incoming job (pq.m = `block`) and becomes `busy`. Meanwhile, a timer (modeled by a `timer` variable) is randomly set to a value from the specified service time interval. Variable `timer` decrements each time step, denoting the residual service time. One time step after the timer reaches 1, the server moves to `block` mode and stays there until the next queue (nq) is `idle`. Then, it returns to `idle` in the subsequent time step.



(a) system structure, (b) behavior of one server.

Figure 5: A closed queueing network.

Suppose each server has a service time anywhere between 1 and 7 steps. The reachable state space grows roughly by a factor of 6.5 with each added queue. As shown in Table 1 (cqn's), the number of recurrence classes is $N + 1$ where N is the number of queues. In fact, under the given service time assumption, one can show that each recurrence class uniquely determines the long-run system behavior for a fixed number of jobs.

The results show that the run time of our method grows polynomially whereas that of the previous method increases exponentially, implying an exponential speed up. The previous method fails to handle systems with more than 6 queues due to insufficient memory while our method can handle systems with many more queues. For example, for a system with 16 queues, our method completes in less than 2 hours.

5 Conclusions

We have presented a BDD-based symbolic method for state classification of discrete-time finite state Markov chains along with the proofs of its correctness. The method iteratively identifies all the transient states and the recurrence classes by the standard reachability analysis. Typically, the number of times the reachability analysis required is very small. Compared with the previous proposed symbolic method, our method has been demonstrated to achieve dramatic improvements on the run time complexity as well as on the memory requirement on almost all examples tested.

References

[1] M. Malhotra and K. S. Trivedi. Power-hierarchy of dependability-model types. *IEEE Trans. on Reliability*, 43(3), Sept. 1994.

[2] O. Daniel Z. Simeu-Abazi and B. Descotes-Genon. Analytical method to evaluate the dependability of manufacturing systems. *Reliability Engineering & Systems Safety*, 55(2), Feb. 1997.

[3] A.M. Stankovic, G.C. Verghese, and D. J. Perreault. Randomized modulation of power converters via Markov chains.

Examples	State space					Run time (CPU sec)	
	Total states	Reachable states	Transient states	Recurrent states	Recurrent classes	Previous method	Our method
dme3cells	1.80×10^{16}	6,579	0	6,579	1	417	7.1
dme6cells	3.25×10^{32}	8.217×10^{6}	0	8.217×10^{6}	1	$8,140^{\dagger}$	123.7
syncarb5	32,768	5,120	0	5,120	1	0.63	0.1
syncarb5	1.07×10^{9}	1.049×10^{7}	0	1.049×10^{7}	1	2.97	0.28
mutex8	524,288	2,048	0	2,048	1	0.48	0.18
periodic	6.87×10^{10}	3,952	0	3,952	1	$1,577^{\dagger}$	1.3
counter8	256	256	0	256	1	0.23	0.18
counter16	65,536	65,536	0	65,536	1	0.43	100.9
pci-2-1	4.29×10^{9}	14,556	0	14,556	1	$3,514^{\dagger}$	14.7
pci-4-1	2.88×10^{17}	1.76×10^{7}	0	1.76×10^{7}	1	$34,620^{\dagger}$	116.5
pci-8-1	1.30×10^{33}	5.24×10^{13}	0	5.24×10^{13}	1	$11h^{\ddagger}$	426.3
diffeq-est	4.29×10^{9}	7,632	2,762	4,870	1	$3,180^{\dagger}$	4.0
diffeq-bka	2.75×10^{11}	11,036	6,176	4,860	1	$1,059^{\dagger}$	6.5
fifo2	65,536	98	2	96	1	8.0	0.23
fifo4	1.68×10^{7}	1,150	18	1,132	1	72.2	0.58
fifo8	1.10×10^{12}	171,518	730	170,788	1	$1,535^{\dagger}$	2.9
fifo16	4.72×10^{21}	4.258×10^{9}	2.48×10^{6}	4.256×10^{9}	1	$11h^{\ddagger}$	24.2
fifo32	5.44×10^{39}	7.736×10^{17}	2.88×10^{13}	7.736×10^{17}	1	$11h^{\ddagger}$	277.0
cqn2	1,024	21	3	18	3	0.45	0.1
cqn4	1.05×10^{6}	1,693	15	1,678	5	24.7	1.7
cqn6	1.07×10^{9}	80,874	63	80,811	7	1,097	14.3
cqn8	1.10×10^{12}	3.083×10^{6}	255	3.083×10^{6}	9	8,138†	60.9
cqn10	1.26×10^{15}	1.328×10^{8}	1,043	1.328×10^{8}	11	3,776†	230.8
cqn12	1.15×10^{18}	5.614×10^{9}	5,535	5.614×10^{9}	13	1,808†	573.0

Table 1: Experimental results: † out of memory, ‡ incomplete.

IEEE Trans. on Control Systems Technology, 5(1), Jan. 1997.

[4] R. Marculescu, D. Marculescu, and M. Pedram. Efficient power estimation for highly correlated input streams. In *Proc. ACM/IEEE Design Automation Conf.*, 1995.

[5] P. A. Beerel and H. S. Wadekar. Energy estimation of speed-independent control circuits. *IEEE Trans. on Computer-Aided Design*, pp. 672–680, June 1996.

[6] M. Holiday and M. Vernon. A generalized timed Petri net model for performance analysis. *IEEE Trans. on Software Engineering*, 13:1297–1310, Dec. 1987.

[7] P. Kudva, G. Gopalakrishnan, E. Brunvand, and V. Akella. Performance analysis and optimization of asynchronous circuits. In *Proc. Int. Conf. Computer Design (ICCD)*, pp. 221–225, Oct. 1994.

[8] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of asynchronous systems based on average time separations of events. In *Proc. Int. Symp. on Adv. Res. in Asynchronous Circuits and Systems*, pp. 64–75, 1997.

[9] R. M. Izquierdo and D. S. Reeves. MPEG VBR slice layer model using linear predictive coding and generalized periodic Markov chains. In *IEEE Int. Performance, Computing and Communications Conf.*, 1997.

[10] T. Takine, T. Suda, and T. Hasegawa. Cell-loss and output process analysis of a finite-buff discrete-time atm queueing system with correlated arrival. *IEEE Trans. on Communications*, 43(2–4), Feb.-March 1995.

[11] G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Trans. on Computer-Aided Design*, pp. 1479–1493, Dec. 1996.

[12] R. E. Bryant. Graph-based algorithm for boolean function munipulation. *IEEE Trans. on Computers*, 35, Aug. 1986.

[13] Y. Matsunaga, P. C. McGeer, and R. K. Brayton. On computing the transitive closure of a state transition relation. In *Proc. ACM/IEEE Design Automation Conf.*, 1993.

[14] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. Springer, 1976.

[15] R. A. Howard. *Dynamic Probabilistic Systems, Volume I: Markov Models*. John Wiley & Sons, Inc, 1971.

[16] G. D. Hachtel and F. Somenzi. A symbolic algorithm for maximum flow in 0-1 networks. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 1993.

[17] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer-Aided Design*, 13(4), Apr. 1994.

[18] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification'90*. American Mathematical Society, June 1990.

[19] R. I. Bahar, E. A. Frohm, C. M. Gaona, and G. D. Hachtel. Algebraic decision diagrams and their applications. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, 1993.

[20] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[21] K. Y. Yun and R. P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proc. Int. Conf. Computer Design (ICCD)*, pp. 118–123. Oct. 1996.

[22] L. Kleinrock. *Queueing Systems, Volume II: Application*. Wiley, New York, 1975.