Memory Efficient State-Space Analysis in Software Model-Checking

Anshuman Mukherjee **Zahir Tari** **Peter Bertok**

School of CS & IT, RMIT University
GPO Box 2476V, Melbourne, VIC 3001, Australia
Email: {amukherj,zahirt,pbertok}@cs.rmit.edu.au

Abstract

Formal methods have an unprecedented ability to endorse the correctness of a system. In spite of that, it has been limited to safety-critical and mission-critical systems owing to significant time and memory costs involved. Lately, our ever increasing dependency on software in all walks of our life has necessitated using formal methods for a wider range of softwares. In this paper, we propose an algorithm to make this possible by reducing the memory requirement for *model checking*, a widely used formal method. A model-checker stores all explored states in memory to ensure termination. The proposed algorithm slash memory costs by storing these states in compressed form. In compressed form, a state is stored as how different it is from its previous state. Our experiments report a memory reduction of 95% with only doubling of computation delay. Aforesaid reduction allows model checking in a machine with only a fraction of memory needed otherwise. Consequently the advantage is twofold, 1)enormous savings as only a small physical memory is required and 2)as more states can now be stored in a memory of same size, the chances of complete state-space analysis is exceedingly high.

Keywords: State-space compression, Model checking, Formal methods, State-space explosion

1 Introduction

Traditionally, a software is considered "fail-safe" if it has passed a rigorous testing phase(Beizer 1990). However, the crash of Ariane 5 launcher(Clarke et al. 2000) and the deaths due to malfunctioning of Therac-25 radiation therapy machine(Rushby 1989) in spite of rigorous software testing suggest otherwise. The team investigating these accidents recommended(Clarke et al. 2000, Rushby 1989) using formal methods(FM) to complement testing as the former assures exhaustive verification of a system. However, FM have a high price-tag attached due to *state space explosion*(Christensen et al. 2001) problem, which compel developers to completely skip FM in order to meet software budget. Considering our dependence on software in everyday life(e.g. traffic signals, elevators), skipping FM amounts to risking millions of human lives. In this paper, we propose methods of reducing the cost of FM so that they are more widely used.

Model Checking(MC)(Clarke et al. 2000) is a formal method which verifies a concurrent system automatically. A model checker requires formal design of the system and the properties to be verified. It then explores the state-space of the system to find a state[1] which violates the given properties, where state-space is the set of states reachable from initial state. If a violating state is found, it is returned as a counterexample. Otherwise, the model checker returns 'yes', implying that the properties are satisfied by all reachable states of the system.

During state-space exploration, there might be states generated more than once. To prevent analysing the same states repeatedly for desired properties, it is necessary to remember the states already explored by storing them in memory. This also ensures termination, a condition where no new states are generated. However, model checking is plagued with state-space explosion problem, often resulting in gigantic number of states. This causes manyfold increase in memory costs, as each new state has to be stored. Such bottlenecks in available memory hinders model checking.

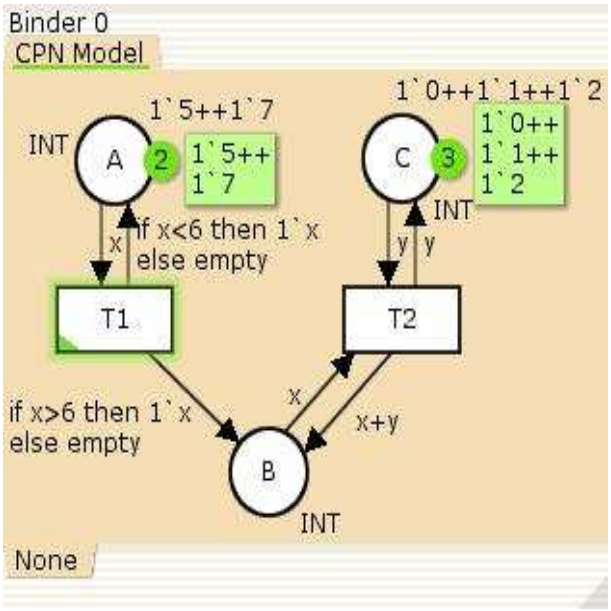
Some solutions based on 'Partial storage' address the problem by storing only a subset of explored states. Although this reduces memory requirement, it is difficult to decide the set of states to be deleted. If a deleted state is reached again in future, it is treated as a new state and explored further. The proposed solution has no such issues as it uses 'Exhaustive storage' in which all explored states are compressed and stored in a suitable data structure(e.g. hash-table). The states need to be decompressed before comparison as it is possible for more than one state to have similar compressed state.

In this paper, we devise a novel method to reduce the memory costs otherwise involved in model checking. We propose storing states in difference form, instead of explicit form, resulting in reduced memory requirements. In difference form, a state is stored as how different it is from its previous states. We propose an algorithm, known as **Sequential algorithm**, to explore the state-space by storing states in difference form.

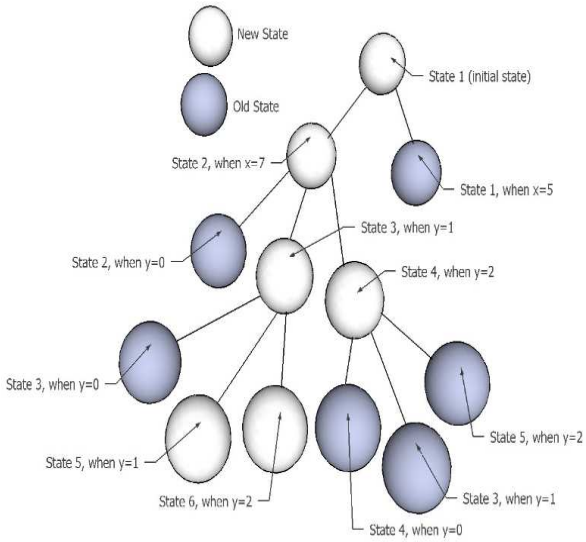
Our contributions can be summarised as:

1. We propose an algorithm to reduce the memory requirement for model checking by storing states in compressed form. The results indicate upto 95% reduction in memory requirement.
2. It is possible for many different explicit states to have the same difference state. Therefore, we propose an algorithm to decompress the states before comparison. Our decompression algorithm only doubles the time needed to generate the state-space. This is 33% lower than the time taken by (Evangelista & Pradat-Peyre 2005).

[1] 'Marking' is sometimes used synonymously to a state



(a) A Coloured Petri-Net model. Variables x and y are of type INT



(b) A part of reachability graph for CPN model in Figure 1(a)

Figure 1: A Coloured Petri-net model and its reachability graph.

The remainder of the paper is organised as follows. Section 2 introduces state-space analysis and Coloured Petri-nets. In section 3 the sequential algorithm is proposed . We tabulate and plot the experimental results in section 4 and discuss the outcome in section 5. We look at the related work in section 6 and conclude in section 7.

2 Background

Model checking involves three basic steps: 1)Modeling the system, 2)Specifying the properties to be verified and 3)Verifying the properties in all reachable states of the model. The first step requires creating a formal representation of the system. The representation depends on model checking tool to be used for verification in step 3. Some common languages for system representation are PROMELA for SPIN(*Basic Spin Manual* 2007), C programming language for BLAST(*Blast Manual* 2008) and Coloured Petri-Nets(CPN) for CPN Tools(*CPN Tools* 2009). Due to subtle differences between these representation languages, it is difficult to propose a generic algorithm for memory reduction. In this paper, the proposed algorithm specifically target CPN models. However, we do not claim any advantage in using CPN models. The proposed algorithm is based on the assertion that "Change in a state is smaller than the state itself" and our algorithm will work for all representation languages, as long as this assertion holds. The assertion is valid because systems usually change in many small steps rather than a single large step. Experiments report a 95% reduction in memory, which further endorse our assertion.

We now define CPN and briefly explain state-space analysis using an example.

2.1 Coloured Petri-Nets

Coloured Petri-Nets(Jensen 1997) are Petri-Nets extended with programming constructs. The concurrent constructs of Petri-Nets are supplemented with data-definition and data manipulation constructs of programming languages. CPN is used for design, specification, simulation and verification of systems. In contrast to PN, each token in CPN has an attached data value. The datatype of this value determine the colour of token. All tokens in a place must be of the colour specified by the colour set of that place.

Figure 1(a) shows a CPN model with 3 places(the circles) and 2 transitions(the rectangles). Place A has 2 tokens(indicated by 2 in the circle next to it) `1‘5` and `1‘7`, where `1‘5` implies that there is 1 token with integer value 5. The tokens in a place are listed next to it and they are separated by '`++`' symbol. The text "`1‘0++1‘1++1‘2`" near place C implies that it has 1 token with value 0, 1 token with value 1 and 1 with value 2. Each place also has its colour(or datatype) inscribed next to it. In this model, all places have colour INT and hence they can only have integer tokens.

A place and a transition are connected by an arc. Transition `T1` in Figure 1(a) has an input arc from place A and two output arcs to places A and B. When `T1` executes, it removes tokens from input place A and adds tokens to output places A and B. The tokens removed/added is found by evaluating the arc inscription. As all these arc inscriptions are defined in terms of variable `x`, they can be evaluated by assigning an appropriate value to `x` and this is called *binding of* `x`. The binding of `x` is decided after considering the tokens in input place and the inscription of input arc. In case of `T1`, input place A has tokens 1‘5 and 1‘7 while the input arc inscription is `x`. Hence the only possible values of `x` for which `T1` is enabled are 5 and 7. When `T1` fires with `x` bound to 5, token 1‘5 is removed from A and added back to A. No token is added to place B as the `if` condition is not satisfied. When `T1` fires with `x` bound to 7, token 1‘7 is removed from A and added to place B. The bindings for which `T2` is enabled can be determined identically.

2.2 State-Space Analysis

In this section, we discuss the problem in detail and outline the proposed solution. State-space analysis of a model is done by generating a reachability graph. Each model has a unique initial state and this is represented by the root node of a reachability graph. At its initial state, the system might have a set of enabled

events which can bring in a change in state. Each of these events are represented by a separate edge from the root node of the reachability graph and lead to a new node representing the new state. These new states are then analysed for the set of enabled events. For each enabled event, an outgoing edge is added to the corresponding node. This in turn generates another set of new states to be analysed identically. This analysis continues till the set of new states have no enabled event.

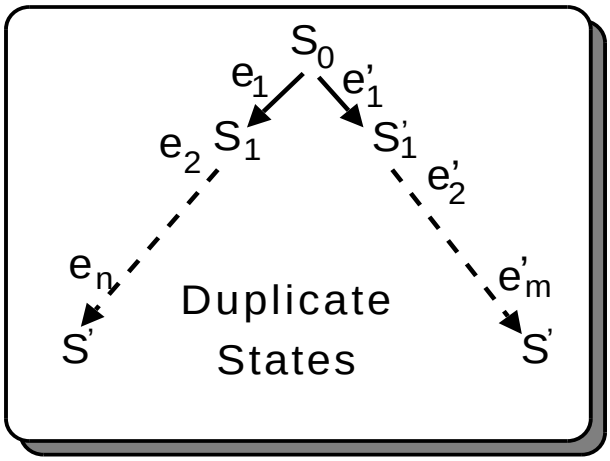


Figure 2: S′ reached using two different sequential of events from S_0

However, it might be possible to reach a state from the initial state by executing different sequence of events. Suppose S_0 is the initial state of a model M and let S′ be a state reached by the following two sequence of events

$$S_0[e_1\rangle S_1[e_2\rangle S_2[e_3\rangle \cdots [e_m\rangle S'$$
$$S_0[e'_1\rangle S'_1[e'_2\rangle S'_2[e'_3\rangle \cdots [e'_n\rangle S'$$

where $\forall i\in[1,m]$: e_i and $\forall j\in[1,n]$: e'_j are events and $S_{i-1}[e_i\rangle S_i$ denotes that event e_i in state S_{i-1} leads to state S_i. This is shown in Figure 2. If $\exists i\in[1,n]:(i<m) \wedge (e_i \neq e'_i)$, the state S′ can be reached using two different sequence of events and therefore it has a *duplicate state*. The reachability graph for M will have two nodes representing the same state S′. However, analysing both the nodes and their children (each of which will also have a duplicate node) is a waste of resources. Larger the number of duplicate nodes for a state, greater the wastage in resources. Furthermore, if there exists a non-empty sequence of events $[e_1e_2\cdots e_r\rangle:r>0$ that cause no net change in state of a model M, the model checker might never terminate. This is shown in Figure 3. Let S be some state of model M and $\forall i\in[1,r]$: e_i be events such that

$$S[e_1\rangle S_1[e_2\rangle S_2[e_3\rangle \cdots [e_r\rangle S$$
$$\text{or } S[e_1e_2e_3\cdots e_r\rangle S$$

Such state of affairs would lead to analysis of the set of states $\{S, S_1, S_2, \cdots, S_{r-1}\}$ forever and state-space analysis might never finish. Consequently, it is necessary to remember the states already explored and ignore any duplicate states encountered. A model-checker remembers explored states by storing them in memory. When a state is generated during state-space exploration, it is compared with the stored states to determine if it is new or duplicate of a previously generated state. If it is a duplicate state, the corresponding node in reachability graph becomes a terminal node and it is not analysed any further. Otherwise, the new state is stored in memory and is analysed for enabled events. However, due to state-space explosion, large amount of memory is needed to store all unique states. In this paper, we propose an algorithm to reduce the memory requirement by storing a state as how different it is from its previous state.

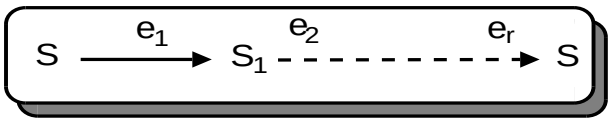


Figure 3: The sequence of events $[e_1e_2\cdots e_r\rangle$ causes no net change in state

At any time, there might be thousands of explored states stored in memory. Comparing each state generated with all stored states might take long. Hence the states are stored in a hash-table to ensure constant time lookup.

We illustrate the problem using an example. Figure 1 shows a Coloured Petri-Net model and a part of its reachability graph. All duplicate nodes in Figure 1(b) are shaded. Initially, the CPN model has 2 tokens in place A and 3 in C. This is represented by State 1 in Figure 1(b) and being the root node of reachability graph, it is stored in memory. The enabled events at this state are (`T1`,`x=5`) and (`T1`,`x=7`), where `T1` is the enabled transition and `x`=5 or 7 is the binding for which it is enabled. Corresponding to these two enabled events, the root node in Figure 1(b) has two outgoing edges, one for each event. When `T1` fires with `x`=5, there is no change in state as all tokens remain in their previous places and the edge corresponding to this event leads to a shaded node in Figure 1(b). As this node represents a duplicate state, it is not analysed any further. The other event (`T1`,`x=7`) results in moving a token from A to B, leading to State 2. Being a new state, it is represented using a bright node in Figure 1(b). Furthermore, it is stored and further analysed for enabled transitions. State 2 has three enabled events: (`T2`,`y=0`), (`T2`,`y=1`) and (`T2`,`y=2`). The first event causes no change in state and is represented by a shaded node in Figure 1(b). The other two events change the value of token in B leading to State 3 and State 4 and these are represented by bright nodes in Figure 1(b). Being new states, they are stored in memory and further analysed for enabled events. Remaining states are explored analogously to generate the complete reachability graph.

The reachability graph in this example has infinite number of states. Other models might have finite number of states. However, the number of states is almost always gigantic leading to state-space explosion(Clarke & Berezin 1998). Complete state-space analysis is possible only when the available memory(α_A) in a machine is at least equal to memory needed to store all unique states in reachability graph(α_M) of model M. Otherwise, if $\alpha_A < \alpha_M$, only a partial state-space analysis can be performed and the analysis stops when memory is full. We propose an algorithm for compact representation and storage of a state. Using sequential algorithm, the memory needed to store all unique states in reachability graph of a model M reduces to α'_M. This allows 1)Complete reachability analysis in a machine with a smaller memory α'_A if $\alpha'_A \geq \alpha'_M$, where $\alpha'_A < \alpha_A$ and $\alpha'_M < \alpha_M$ 2)Even when $\alpha_A < \alpha'_M$, the partial state-space analysis can have at least a few more steps. With available memory remaining same, we are able to create a reachability graph with more states due to less memory needed to store a state.

3 Proposed Algorithm for Memory Efficient State-Space Analysis

In this section, sequential algorithm is proposed for memory efficient state-space analysis. The algorithm proposed in this section specifically target CPN models. However, we do not claim of any advantage in using CPN models. As stated previously, the proposed algorithm is based on assertion that "Change in a state is smaller than the state itself" and as long as this assertion holds, the proposed algorithm is valid for all modeling languages. For CPN, a change in state occurs if one or more tokens either 1)move to another place, 2)change their value, 3)are created in the model, 4)are deleted from the model or a combination of these such that the colour of each token match the colour-set of containing place. In a CPN model, these changes are brought in by a transition. However, a transition usually modifies the place and value information of only a small number of tokens. Furthermore, a very small number of tokens are usually created or deleted by a transition. For example transition `T1` in Figure 1(a) fires with `x=7`, it only changes the place of token 1‘7. Therefore it is substantially cheaper to store a state S as how different it is from its previous state. Based on this, we propose sequential algorithm to generate memory efficient reachability graph in next section.

3.1 Sequential Algorithm

In this section, we introduce sequential algorithm to reduce the memory requirements of model checking. Such a reduction will increase the affordability and consequent use of model checking in software development. The focus of sequential algorithm is storing states in difference form which is defined as:

Definition 1 *The difference form of a state S_{st}, with previous state S_{pv}, is defined as the changes necessary in S_{pv} to generate S_{st} and it is denoted as D_{st}. If D_{st} is the difference form of a state and S_{pv} is its previous state, the state S_{st} can be regenerated in explicit form as S_{st}=S_{pv}+D_{st}.*

An explicit state has information for all tokens and is often referred as state in this paper, omitting the adjective 'explicit'. We illustrate how to find the difference form of a state using an example.

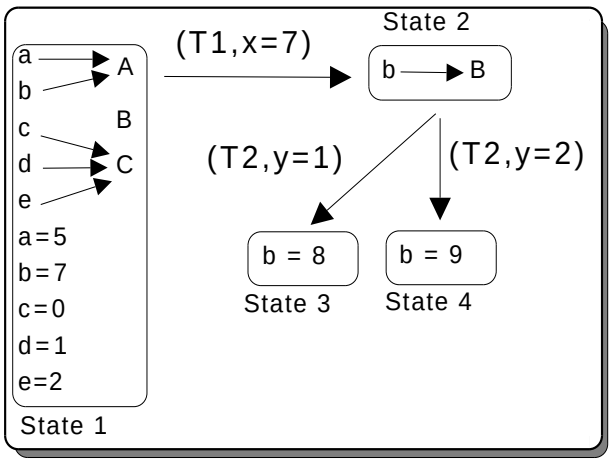
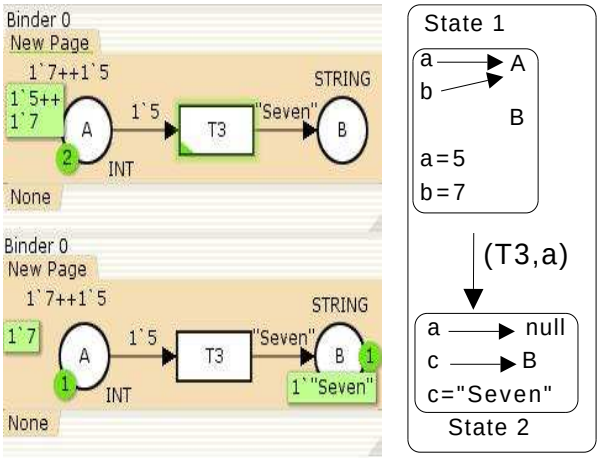


Figure 4: A part of reachability graph in Figure 1(b) using sequential algorithm

Figure 4 presents a portion of reachability graph for the CPN model in Figure 1(a) using sequential algorithm. Initially the model is in State 1 and since it does not have a previous state, it is stored in explicit form in Figure 4. Each token in the model is given a name in Figure 4. Furthermore, its place is assigned by an arrow(→) and value is assigned by an equal(=) symbol. For example, the token 1'5 in place A of Figure 1(a) is named `a`. Its place is assigned as `a`→A while the value is assigned as `a`=5. Similarly, token 1'7 in A is named `b` and assigned place and value as `b`→A and `b`=7. All other tokens are named arbitrarily and assigned place and value accordingly.



(a) A CPN model where T3 fires to delete token 1'5 and create token 1'"Seven"

(b) Reachability graph using sequential algorithm

Figure 5: State change when tokens are created and/or deleted

When the event (`T1`,`x=5`) occurs, the model persist in State 1. As a result, the difference form is empty(or null) and not drawn in Figure 4. However, (`T1`,`x=7`) takes it to State 2. In order to store the new state in difference form, we need to find the changes in State 1 brought by this event. We find that the event moved token 1‘7 to place B. This information is sufficient to construct State 2 from State 1. We therefore store State 2 in difference form as `b`→B.

The event (`T2`,`y=1`) in State 2 lead to State 3. Likewise, the event (`T2`,`y=2`) lead to State 4. In order to store these new states in difference form, the changes in State 2 brought by these events need to be found. On inspecting these events, both are found to change the value of token in place B. While (`T2`,`y=1`) changes the value to 8, (`T2`,`y=9`) changes it to 9. Given State 2 in explicit form, this information is sufficient to construct State 3 and State 4. Accordingly, State 3 is stored as `b`=8 while State 4 is stored as `b`=9. The difference form for other states are calculated identically. Additionally, each state also store a pointer to its previous state. This is necessary to regenerate the states as explained later. As evident from this example, it takes less space to store states in difference form.

A state change also occurs when an event creates or deletes one or more tokens. If an event deletes a token `a`, the new state can be represented in difference form by assigning the place for `a` as `null`(`a`→`null`). Similarly when an event creates a new token, it is given an arbitrary name and assigned the corresponding place and value information. This is illustrated by an example in Figure 5. The CPN model in Figure 5(a) has a transition `T3` which removes token `1‘5` from place A and adds `1‘"Seven"` to place B. Considering the value in latter token, place B is assigned colour-set STRING. The reachability graph of the model using sequential algorithm is presented in Figure 5(b). The tokens in place A are assigned names `a`

and b. Initial state of the model is stored explicitly in Figure 5(b) as there is no previous state to calculate difference. When event (T3,1'5) occurs, it deletes the token a and creates a new token which we name c. The new state can be stored in difference form by assigning the place of a to null and assigning the place and value information for newly created token. Given State 1 in explicit form, aforesaid information is sufficient to regenerate State 2. This example further endorse a reduction in memory requirement when states are stored in difference form.

In this section, we explained how to obtain the difference form of a state and demonstrated the memory reduction when states are stored in difference form. However, more than one explicit state can produce the same difference state. This necessitates converting states into explicit form before comparison. This is explained with an example in next section.

3.1.1 Expanding a State in Difference Form

In this section, we demonstrate backtracking in order to revert a difference state. This is necessary for comparison as more than one explicit state can produce the same difference state.

When a state is generated during state-space exploration, it has to be compared with stored states to determine if it is new. However, compressing and comparing it with states stored in difference form might lead to an error owing to multiple states having the same difference form. Supposing three states S_a, S_b and S_c produce the same difference form D_{abc}. When either of the three states is encountered for the first time, D_{abc} is stored in memory. When the other two states are encountered and compared with stored states in compressed form, they are wrongly interpreted as duplicate state. Therefore it is essential to revert a stored state before comparing. Such a conversion is called *expanding* and is done by *backtracking*.

Definition 2 *Backtracking is the process of regenerating a state by recursively adding the most recent changes for each token to its previous state until a state stored in explicit form is reached. If S_n and D_n are the explicit and difference states at depth n of a reachability graph, the former can be obtained from latter using a backtracking function BK, where S_n=$BK(D_n)$=D_n+S_{n-1}. Since S_0 is always in explicit form, this equation can be solved for all $n \leq h$, where h is the height of the reachability graph.*

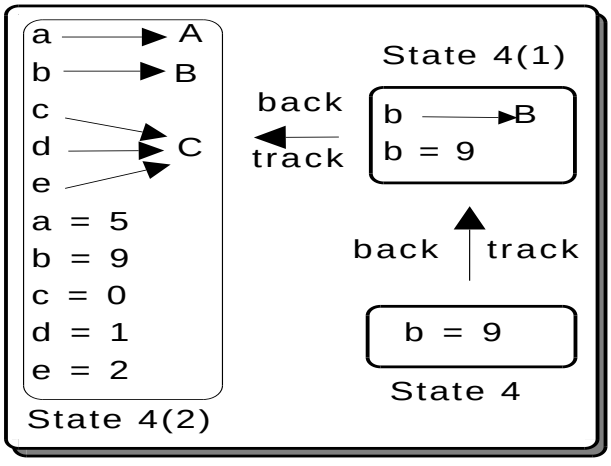


Figure 6: Backtracking to expand State 4 in Figure 4

We illustrate backtracking with an example. State 4 is stored in difference form in Figure 4 and in order to expand it, we need to backtrack till an explicit state is encountered, as shown in Figure 6. Initially, State 4 contains new value for token b and from Definition 1, we should get State 4 in explicit form by updating its previous state(which we hope is in explicit form) with this value. However, on backtracking one step in Figure 4, we reach State 2 which is also stored in difference form. But it gives additional information about the place of token b. We add the place information from State 2 with value information from State 4 and get a meta-state 4(1) shown in Figure 6. We call 4(1) a meta-state as it was obtained by combining two different states. Using Definition 2, State 4 can now be obtained by updating the previous state of State 2 with information in metastate 4(1). On further backtracking, State 1 is encountered which is actually in expanded form. We update it with the information in 4(1) and get another metastate 4(2). By Definition 2, 4(2) is State 4 in expanded form.

As backtracking is an additional overhead when states are stored in difference form, they increase the time needed for state-space analysis. Definition 2 requires backtracking to initial state S_0 for expanding each state. However, this leads to large delay with increase in height of reachability graph. In the next section, we discuss ways for reducing this problem.

3.1.2 Decreasing the Cost of Expanding

In this section, we discuss ways of reducing the additional delay incurred while backtracking. Reducing this delay will reduce the overall time for model checking.

So far we have stored only the initial state in explicit form while all other states to be stored in difference form. Although this ensures maximum reduction in memory requirement, Definition 2 will require backtracking to the initial state for expanding. As a result, the states far from initial state take long to expand.

In order to reduce the delay, the number of backtracking steps need to be minimised. If every state at depth $i\delta$:$i \in N^2$ steps from initial state is stored in expanded form, expanding a state will never need a backtracking greater than δ-1 steps. Therefore starting from initial state, δ^{th}, $2\delta^{th}$, $3\delta^{th} \cdots$ states are stored in expanded form. The algorithm can be tuned by accepting different values of δ. Finally, we propose our algorithm next section.

3.1.3 Proposed Algorithm

In this section, the proposed algorithm is introduced and explained. As specified previously, sequential algorithm achieves memory reduction by storing states in difference form. Starting from the initial state, it explores remaining reachable states of the model using depth-first search(DFS) algorithm(Cormen et al. 2001). Each explored state is stored in a hash-table.

When a state is generated, a hash-function is used to find its index in hash table. If this index is empty, the state is new. Its difference form is calculated and inserted at this index. Furthermore, the enabled transitions are identified and one of them is fired. Otherwise, if there are states already stored at this index, they are all expanded and compared with the generated state. If there is no match, the state is new and it is inserted at the head of list at this index. Additionally, one of its enabled transitions is executed. In case of a match, the state is duplicate of a state analysed previously. It is neither stored nor analysed for enabled transitions.

[2] N is the set of natural numbers starting from 0

The proposed algorithm calculates the difference form of a state by comparing it with its previous state. However to reduce delay in backtracking, all states at depth iδ:i$\in$N from initial state are stored in explicit form, where δ is the shortest distance between two explicit states. In order to expand a state, the algorithm implements backtracking until an explicit state is encountered.

The proposed algorithm also implements two-level hashing at an index if the number of states stored at that index exceeds a threshold. In our algorithm, we set threshold as M/10, where M is an estimation of the total number of reachable states. When two level hashing is used, the index of primary hash table contains hash-function for secondary hash table.

The proposed sequential algorithm has three parts:

1. `SEARCH:` The steps are listed in Algorithm 1. This algorithm accepts a state(S_{st}), it's previous state(S_{pv}) and the distance of S_{st} from last expanded state(depth) as input. A hash function H is used to find the index for state S_{st} as shown in step 1. The algorithm then checks the content of hash-table at this index. There can be three possibilities.

Algorithm 1: SEARCH (State S_{st}, int depth, State S_{pv})

```
   Data: current state S_st,steps away from last
         explicit state(depth), previous state S_pv
   Result: Decide if a state generated is new
 1 i←H[S_st] ;
 2 if HASH[i]=NULL then
 3 |  INSERT(S_st,depth,S_pv);
 4 |  foreach S' such that S_st[(t,c)⟩S' do
   |  SEARCH(S',(depth+1)mod δ,S_st);
 5 else if HASH[i] points to a linked list then
 6 |  foreach state D in linked list do
 7 |  |  if D is in difference form then
   |  |  D←RECONSTRUCT(D);
 8 |  |  if D=S_st then return;
 9 |  end
10 |  INSERT(S_st,depth,S_pv);
11 |  foreach S' such that S_st[(t,c)⟩S' do
   |  SEARCH(S',(depth+1)mod δ,S_st);
12 else if HASH[i] contains a hash function then
13 |  H' ← HASH[i];
14 |  j←H'[S_st] ;
15 |  if HASHi [j] is empty then
16 |  |  INSERT(S_st,depth,S_pv);
17 |  |  foreach S' such that S_st[(t,c)⟩S' do
   |  |  SEARCH(S',(depth+1)mod δ,S_st);
18 |  else
19 |  |  if HASHi[j] is in difference form then
20 |  |  |  HASHi[j]←RECONSTRUCT(HASHi[j]);
21 |  |  end
22 |  |  if HASHi[j]=S_st then return;
23 |  |  INSERT(S_st,depth,S_pv);
24 |  |  foreach S' such that S_st[(t,c)⟩S' do
   |  |  SEARCH(S',(depth+1)mod δ,S_st);
25 |  end
26 end
```

   (a) *Hash table contain NULL at this index:* In this case, it is the first time this state is generated. Hence Algorithm 2 is called to store the state at this index. Any enabled event at this state is fired. Steps 2-4 in Algorithm 1 check and handle this case.

   (b) *Hash table contain a linked-list at this index:* In this case, each state in the linked-list has hashed to this index. S_{st} is compared with each state in this list. If a state is stored in difference form, it is expanded before comparison using Algorithm 3. In case of a match, the state is neither stored nor analysed for an enabled event. Otherwise, the state is stored at the head of linked list using Algorithm 2 and an enabled event is fired. Steps 5-11 in Algorithm 1 check and handle this case.

   (c) *Hash table contain a hash-function at this index:* If this case, all states which hashed to this index are stored in a separate hash-table HASHi indexed by the function H′ stored at this index. In step 14, the index in second hash table is calculated using this hash function. Step 15 checks if this index is empty or has a state stored. In case this index is empty or does not contain this state, it is inserted at this index using Algorithm 2 and its enabled events are fired. Otherwise the algorithm returns. Steps 15-24 in Algorithm 1 handle these cases.

Algorithm 2: INSERT(State S_{st}, int depth, State S_{pv})

```
   Data: current state S_st,steps away from last
         explicit state(depth), previous state S_pv
   Result: Insert state S_st into hash table
 1 if depth=0 then
 2 |  new.type←explicit;
 3 |  new.state←S_st;
 4 else
 5 |  new.type←difference;
 6 |  new.state←S_st-S_pv;
 7 |  new.prev←S_pv;
 8 end
 9 i← H[S_st] ;
10 if HASH[i]=NULL then
11 |  HASH[i]=new;
12 else if HASH[i] points to a linked list then
13 |  insert new at the head of linked list;
14 |  if length(linked list)≥ |M|/10 then
15 |  |  foreach state d in linked list do
16 |  |  |  if d is in difference form then
17 |  |  |  |  d←RECONSTRUCT(d);
18 |  |  |  end
19 |  |  |  add d to HASHi[H'[d]];
20 |  |  end
21 |  |  HASH[i]←H'
22 |  end
23 else if HASH[i] points to a hash function then
24 |  H' ← HASH[i];
25 |  j←H'[S_st] ;
26 |  HASHi[j]←new;
27 end
```

2. `INSERT:` This algorithm is responsible for inserting a state into hash table and is listed in Algorithm 2. It accepts a state(S_{st}), it's previous state(S_{pv}) and the distance of S_{st} from last expanded state(depth) as input. The fields of a pointer "new" are assigned the required values before storing it in appropriate index. Based on the value of delta, the state is either stored in explicit or difference form and this is assigned to 'type' field of pointer 'new'. In case of former, the explicit state S_{st} is assigned to 'state' field of

'new'. Otherwise the difference, given by "S_{st}-S_{pv}" is assigned to 'state' field. Additionally, in latter case, a pointer to previous state is stored in 'prev' field of 'new'. This is shown in steps 1-8 of Algorithm 2.

The index of hash-table at which this state is to be stored is calculated in step 9. There could be three possible cases:

(a) *Hash table contain NULL at this index:* This is the case when S_{pv} is generated for the first time. The contents of pointer 'new' is simply copied to this index of hash-table. This is shown in steps 10-11 of Algorithm 2.

(b) *Hash table contain a linked-list at this index:* In this case, the contents of 'new' is copied to head of linked list. Furthermore, it is checked if the list contains more than 10% of an estimated total number of states. In that case, the states in this linked list is stored in another hash table and the hash function is stored at this index. This is done in steps 12-22 of Algorithm 2.

(c) *Hash table contain a hash-function at this index:* In this case, the hash-function stored at this index is used to find the index in secondary hash-table and the contents of pointer 'new' is copied to that index. Steps 23-26 in Algorithm 2 handle this case.

3. `RECONSTRUCT:` This algorithm accepts a difference state and expands it by backtracking. The steps are listed in Algorithm 3. In steps 1-4, the algorithm backtracks and add each state encountered until an explicit state is reached. Finally, the state D_{st} in explicit form can be calculated by adding the sum to the explicit state encountered. This is shown in step 5.

Algorithm 3: RECONSTRUCT(State D_{st})

```
   Data: State D_st in difference form
   Result: Expanded form of D is returned
1  while d.type=difference do
2  |   sum=sum+d.state ;
3  |   d=d.prev;
4  end
5  return d+sum;
```

3.1.4 Complexity Analysis

The proposed sequential algorithm promises to reduce the amount of space necessary to store the states by using difference states. However, this reduction is accompanied by a delay due to backtracking. In this section, we calculate the reduction provided and derive the time needed for extra processing.

Let δ be the distance between two expanded states. We pointed out earlier that the initial state is stored in expanded form. Other states in expanded form are those at depth δ, 2δ, and so on. If a reachability graph has height n, the depth of last expanded node is $\lfloor \frac{n}{\delta} \rfloor * \delta$.

Assuming that the average number of new states generated by a transition is k(>1), the number of states at depth d is given by k^d and this is illustrated in Figure 7. All dark circled represent explicit states while shaded circles represent difference states. The number of expanded states in a reachability of height n is the sum of the number of expanded states at depth 0, δ, 2δ, $\cdots$, $\lfloor \frac{n}{\delta} \rfloor * \delta$. This is given by

$$\beta_{expanded} = k^0 + k^\delta + k^{2\delta} + \cdots + k^{\lfloor \frac{n}{\delta} \rfloor * \delta}$$

This is a geometric progression (Bronshtein et al. 1997) with initial term a=1 and ratio r=k^δ. Hence, the sum is given by

$$\beta_{expanded} = \frac{a(r^{n+1}-1)}{r-1} = \frac{k^{(\lfloor \frac{n}{\delta} \rfloor+1)*\delta}-1}{k^\delta-1} \quad (1)$$

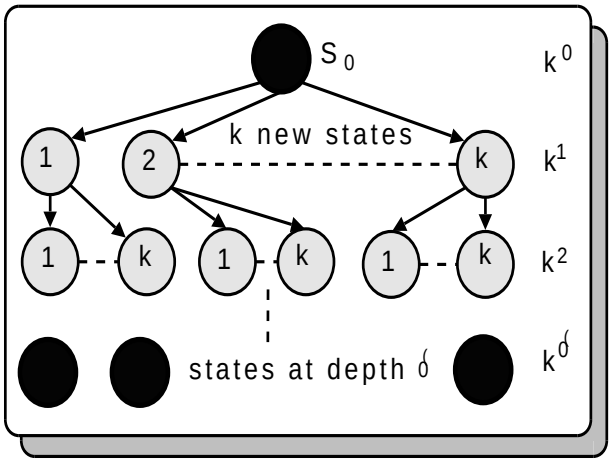


Figure 7: At depth d, the number of states is k^d. All states at depth δ are explicit

Similarly, the total number of states is another geometric progression with initial term a=1 and ratio r=k.

$$\beta_{total} = k^0 + k^1 + k^2 + \cdots + k^n$$
$$\text{or } \beta_{total} = \frac{a(r^{n+1}-1)}{r-1} = \frac{k^{n+1}-1}{k-1} \quad (2)$$

Therefore the number of states in difference form is given by

$$\beta_{difference} = \beta_{total} - \beta_{expanded}$$

Assigning $\beta_{expanded}$ from equation 1 and β_{total} from equation 2 we get

$$\beta_{difference} = \frac{k^{n+1}-1}{k-1} - \frac{k^{(\lfloor \frac{n}{\delta} \rfloor+1)*\delta}-1}{k^\delta-1}$$

Percentage Reduction in Memory: The number of states in difference and explicit forms is given by equations 1 and 2. Suppose the memory occupied by an explicit state is λ, while a state stored in difference form occupies $x*\lambda$ memory, where $0<x<1$. Therefore the memory needed to generate a reachability graph of depth n without using our algorithm is

$$\Lambda_{withoutalgo} = \beta_{total} * \lambda \quad (3)$$

When using our algorithm, the memory needed to generate the same reachability graph is

$$\Lambda_{withalgo} = \beta_{difference} * \lambda * x + \beta_{expanded} * \lambda \quad (4)$$

The percentage reduction in memory denoted by Δ is

$$\Delta = \frac{\Lambda_{withoutalgo} - \Lambda_{withalgo}}{\Lambda_{withoutalgo}} \quad (5)$$

Using equations 3 and 4 in 5, we get

$$\text{or } \Delta = \frac{\beta_{total}*\lambda - \beta_{difference}*\lambda*x - \beta_{expanded}*\lambda}{\beta_{total}*\lambda}$$

Substituting $\beta_{difference}$ as $\beta_{total} - \beta_{expanded}$

$$\Delta = (1\text{-}x)*\left(1 - \frac{\beta_{expanded}}{\beta_{total}}\right)$$
$$\text{or } \Delta = (1\text{-}x)*\left(1 - \frac{(k^{(\lfloor \frac{n}{\delta} \rfloor+1)*\delta}-1)*(k-1)}{(k^\delta-1)*(k^{n+1}-1)}\right)$$

Time needed for Extra Processing: The extra time required when states are stored in difference form is now calculated. We only consider the delays due to backtracking as any other delay is common for both explicit and difference states.

Let i be an integer between 1 and n. If the height of reachability graph is i, the number of states in explicit and difference forms are given by

$$\beta'_{total} = \frac{k^{i+1}-1}{k-1},\ \beta'_{expanded} = \frac{k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1}{k-1}$$
$$\text{and } \beta'_{difference} = \frac{k^{i+1}-1}{k-1} - \frac{k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1}{k^{\delta}-1}$$

When a state S_{st} is generated, it is compared with the state stored at an index given by the hash-function. The probability that this state is stored in expanded or difference form can be calculated as

$$P_{expanded} = \frac{\beta'_{expanded}}{\beta'_{total}} \text{ and } P_{difference} = \frac{\beta'_{difference}}{\beta'_{total}}$$

If the state is stored in difference form, it has to be first expanded by backtracking and then compared with S_{st}. Hence, time taken for comparing S_{st} with the stored states is given by

$$T_{comparison} = T_{expanded} + T_{difference}$$

Suppose the time for comparing two expanded states is ϵ, while it takes y$*\epsilon$ time for backtracking a single step. In the worst case, a backtracking of $(\delta-1)$ steps is necessary to expand the state. Therefore the time can be calculated as

$$\begin{aligned} T_{comparison} &= P_{expanded} * \epsilon + P_{difference} * \epsilon(\delta-1)y \\ &= \frac{\epsilon(1-y(\delta-1))\beta'_{expand}}{\beta'_{total}} + \epsilon(\delta-1)y \\ &= \frac{\epsilon(k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \epsilon(\delta-1)y \end{aligned}$$

This is the time taken for comparing a state generated with a stored state in difference form. All comparison at a particular depth takes place concurrently. Hence the total time taken to generate reachability graph of height n is the sum of time taken for one comparison at each level. This is denoted by π, where

$$\pi = \sum_{i=0}^{n} \frac{\epsilon(k^{(\lfloor \frac{i}{\delta} \rfloor+1)*\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \epsilon(\delta-1)y$$

Since $\lfloor \frac{n}{\delta} \rfloor$=0 for 0≤i< δ, $\lfloor \frac{n}{\delta} \rfloor$=1 for δ ≤i< 2δ etc.,

$$\begin{aligned} \pi = &\sum_{i=0}^{\delta} \frac{\epsilon(k^{\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \\ &\sum_{i=\delta}^{2\delta} \frac{\epsilon(k^{2\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \cdots + \\ &\sum_{i=z\delta}^{n} \frac{\epsilon(k^{(z+1)\delta}-1)(k-1)(1-y(\delta-1))}{(k^{\delta}-1)(k^{i+1}-1)} + \epsilon(\delta-1)y \end{aligned}$$

where z=$\lfloor \frac{n}{\delta} \rfloor$. This is the time taken to generate a reachability graph of height n when the proposed algorithm is used.

4 Experimental Result

The proposed algorithm was tested on a desktop with 2.8GHz Intel Pentium D processor and 1GB RAM. The desktop had Ubuntu 8.04 desktop version OS installed and our C source code was compiled using GNU C compiler(gcc).

We used six different Coloured Petri-net models to run our experiment. The number of places and tokens in each CPN model is listed in Table 1 and Table 2. If a model had m tokens and n places, each token was assigned an integer name i:i∈[0,m-1] and each place was assigned an integer name j:j∈[0,n-1]. Initially, all tokens were in place 0. At each state, the set of enabled transitions were selected randomly and one of these transitions was fired. This allow having a large number of transitions in a model without specifying the bindings for which they are enabled.

For each CPN model, we have calculated the time and space needed to generate first 500 unique states using sequential algorithm and without using it. Furthermore, sequential algorithm require a non-negative integer value of δ and we have assigned it the set of values {1,2,3,7,20}. When proposed algorithm is not used, we assign 0 to δ. The results are listed in Table 1 and Table 2. δ is the shortest distance between two explicit states.

Table 1 shows the memory needed(given by Λ) to store first 500 states of CPN models used and the percentage reduction in memory requirement(given by Δ) for different values of δ. For each model, the memory requirement is highest either when not using our algorithm(δ=0), or when using it with δ=1. In Figure 8, memory required is plotted against value of δ. On increasing the value of δ, the memory requirement decrease for all models. Furthermore, the decrease is significantly higher for large models, with a significantly greater number of places and tokens, as compared to small models. For instance, the CPN model with 1500 places and 2000 tokens used 95% less space when our algorithm was used with δ=20. Compared to this, the reduction was 76% for a CPN model with 4 places and 5 tokens. Nevertheless, the reduction is massive for models of all size and for all values of δ, as evident from Table 1 and Figure 8.

Table 2 shows the time needed(given by π) to generate first 500 states of CPN models used and the percentage increase in delay(given by η) for different values of δ. For each model, the delay is minimum when our algorithm is not used(δ=0). In Figure 9, delay is plotted against value of δ. When our algorithm is used, delay increase with increase in value of δ. This increase is massive for small models. A model with 4 states and 5 tokens generates 500 states almost instantly(π=0) without using our algorithm. The same model needs 1.5 second when our algorithm is used with δ=20. The percentage increase in delay (η) decrease with an increase in size of model. The model with 1500 places and 2000 tokens has twice the delay when our algorithm is used with δ=20 as compared to when δ=0(algorithm not used). A comparison of results in Table 1 and Table 2 clearly shows that the reduction in memory requirement comes at the cost of extra delay in processing. This is further evident in Figure 10 where the required memory decrease and delay increase with increase in value of δ. However, the memory reduction is massive as compared to the increase in delay, especially for moderate to large size models. A model with 1500 places and 2000 tokens had 95% reduction in memory with double the delay. Due to inherent complexity of most modern systems, their models are almost always large. Our algorithm is addressing a niche for such systems.

5 Discussion

The proposed algorithm reduce the memory requirement for model checking by storing states in difference form and thereby allow model checking in a machine with a fraction of memory needed otherwise. This might lead to wider use of model checking in software verification and subsequent production of reliable software systems. Although there is an increase in delay due to backtracking, the results illustrate that the delay is small when compared to the massive reduction in memory obtained.

Reduction in memory requirement Λ: State-space analysis without using our algorithm will store all states in explicit form, leading to maximum mem-

Table 1: Space occupied(in bytes) by first 500 states of CPN models(Λ) and percentage decrease in space(Δ)

n	m	δ=0	δ=1		δ=2		δ=3		δ=7		δ=20	
		Λ	Λ	Δ	Λ	Δ	Λ	Δ	Λ	Δ	Λ	Δ
4	5	9980	9980	0%	5996	40%	4668	53%	3148	68%	2396	76%
60	90	179640	179640	0%	90996	49%	61448	66%	27628	85%	10896	94%
200	400	798400	798400	0%	400996	50%	268528	66%	116908	85%	41896	95%
400	700	1397200	1397200	0%	700996	50%	468928	66%	203308	85%	71896	95%
800	1000	1996000	1996000	0%	1000996	50%	669328	66%	289708	85%	101896	95%
1500	2000	3992000	3992000	0%	2000996	50%	1337328	67%	577708	86%	201896	95%

Table 2: Time(in msec) to generate first 500 states of CPN models(π) and percentage increase in time(η)

n	m	δ=0	δ=1		δ=2		δ=3		δ=7		δ=20	
		π	π	η	π	η	π	η	π	η	π	η
4	5	$\approx$0	20	∞	90	∞	160	∞	430	∞	1490	∞
60	90	20	40	100%	80	300%	100	400%	210	950%	570	2750%
200	400	80	130	62%	170	112%	200	150%	320	300%	680	750%
400	700	140	220	57%	260	85%	290	107%	410	193%	770	450%
800	1000	200	310	55%	350	75%	380	90%	500	150%	850	325%
1500	2000	400	620	55%	660	65%	680	70%	820	105%	1200	200%

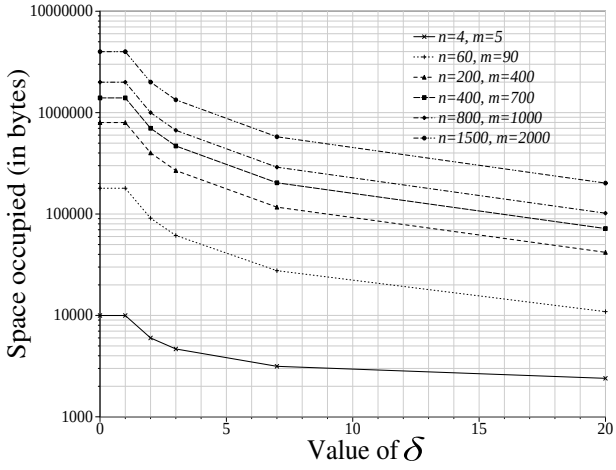


Figure 8: Memory requirement decrease with increase in value of δ. δ=0 means algorithm not used. Y-axis uses log scale.

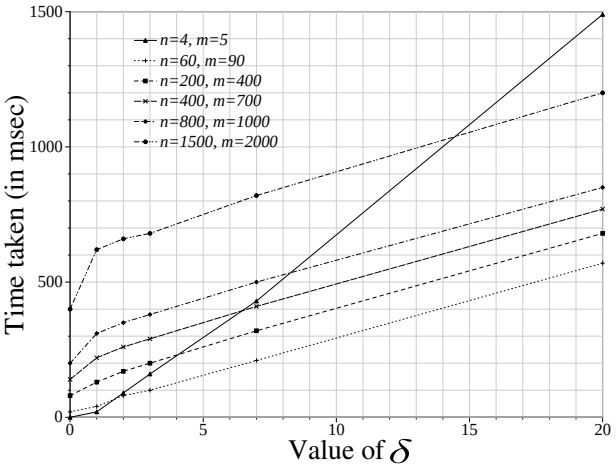


Figure 9: Delay increase with increase in value of δ. δ=0 when algorithm not used.

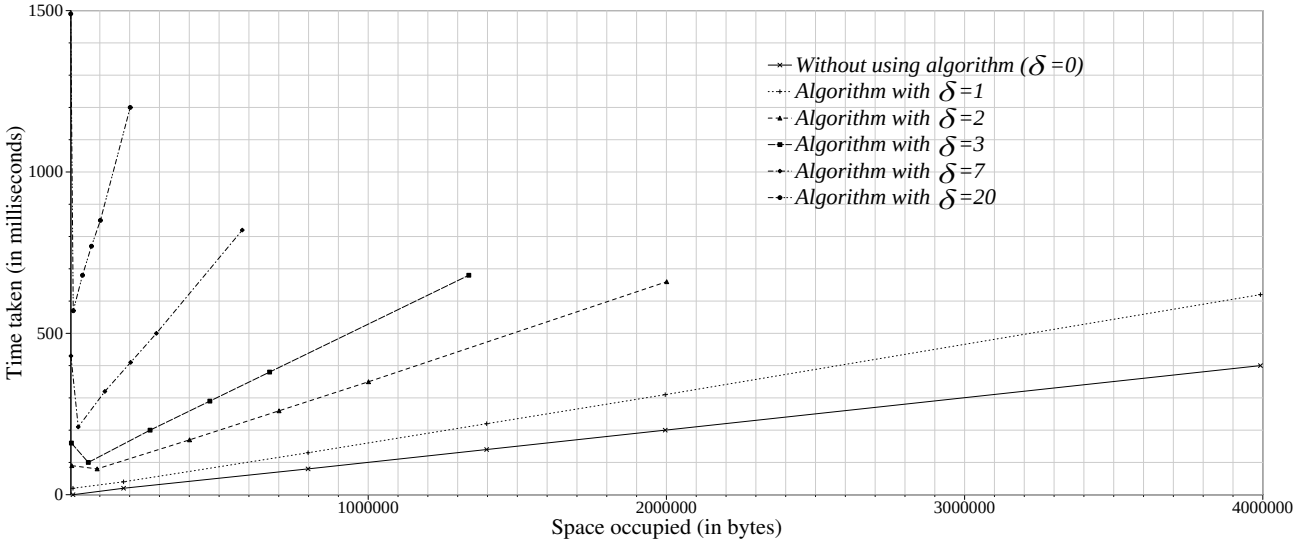


Figure 10: Delay increase and memory requirement decrease with increase in value of δ. Each curve is for a different value of δ as indicated by the legend. The six points on a curve correspond to six CPN models used

ory requirement. This holds for our results in Figure 8. Furthermore, using our algorithm with δ=1 also stores all states in explicit form, keeping Λ unchanged. However when δ=2, every alternate state is stored in explicit form. This leads to almost 50% reduction in Λ as only half the total number of states are in explicit form. Similarly, when δ=3, one in every three states are stored in explicit form leading to 66% reduction in Λ. When δ=7, one in seven states is stored in explicit form leading to 85% reduction in Λ. Finally, when δ=20, one in 20 states is stored in explicit form resulting in 95% reduction in value of Λ.

The reduction for CPN model with 4 places and 5 tokens is low as compared to other models. The reason being that the size of an explicit state is almost same as a difference state for a small model. Therefore, replacing explicit state with difference state do not make a big difference.

Increase in delay π: Two factors contribute to overall delay: 1)backtracking to expand a difference state 2)when state-space is being explored using DFS algorithm and a duplicate state is encountered, the stack is popped till a state with an enabled event(transition) is encountered. Popping a stack is time intensive operation.

A small model has less number of possible states and therefore the chances of encountering a duplicate state is high. The CPN model with 4 places and 5 tokens encountered 469 duplicate states before generating 500^{th} unique state. As compared to this, the model with 60 places and 90 tokens encountered only 1 duplicate state before generating 500^{th} state. The delay in popping stack, combined with backtracking delay lead to large π for small models.

When model-checking, we need not backtrack if all states are in explicit form. This leads to low π when δ=0 or δ=1. However, due to extra processing delay of our algorithm, the delay for δ=1 is higher than δ=0. On further increasing δ, delay increases due to backtracking. Higher the value of δ, more is the backtracking needed to expand a state and greater the delay.

6 Related Work

All solutions proposed to store state-space can be classified as either of 1)Exhaustive storage 2)Partial storage or 3)Lossy storage. The proposed algorithm is based on exhaustive storage, wherein all explored states of a model are compressed and stored in a suitable data structure(e.g. hash-table) to ensure constant time lookup. Table 3 compares proposed algorithm with other solutions based on this approach and the state-space compression they provide. The table also gives the additional delay incurred when using a solution. The proposed algorithm provides reduction equivalent to (Evangelista & Pradat-Peyre 2005) with only 2/3 of its delay.

Table 3: A comparison of solutions based on exhaustive storage

Method	Run-Time	Memory-Use
No Algorithm	100%	100%
(Schmidt 2003)	130%	60%
(Evangelista & Pradat-Peyre 2005)	300%	05%
(Holzmann 1997)	280%	18.3%
Sequential Algorithm proposed	200%	05%

In Partial storage, only a subset of the explored states are stored. Sweep-line method, proposed in (Christensen et al. 2001), is a solution based on partial storage where a state is deleted if it cannot be reached again in future. However, it is difficult to decide the states to be deleted. Furthermore, it is not a generic solution as for different systems, we might need to delete a different set of states.

Lossy storage is similar to exhaustive storage wherein explored states are stored in compressed form in suitable data structure. However, it is not possible to decompress the states. In order to determine if a state is new, it is also compressed and compared with stored states. As pointed out previously, multiple states can have same compressed form. This often results in falsely implicating a state as duplicate. An interesting solution based on lossy storage is proposed in (Wolper et al. 1993)

7 Conclusion

In this paper, we have reduced the memory requirement for model checking by storing states in difference form. Consequently, model checking would acquire a bigger role in verification of a wide range of softwares. This will ensure safety and reliability of software systems used in all walks of life. Experimental results indicate that our algorithm performs remarkably better for large models. Contemporary systems have high level of complexity, often leading to large models. The proposed algorithm is addressing a niche for such systems.

In future, we aim propose a distributed model checking algorithm by extending the sequential algorithm. This will reduce the accompanying delay as a result of parallel processing.

References

Basic Spin Manual (2007).
URL: *http://spinroot.com/spin/Man/Manual.html*

Beizer, B. (1990), *Software testing techniques.*

Blast Manual (2008).
URL: *http://www.cs.sfu.ca/ dbeyer/blast$_d$oc/blast.pdf*

Bronshtein, I. N., Semendyayev, K. A. & Kirsch, K. A. (1997), *Handbook of mathematics (3rd ed.)*, Springer-Verlag, London, UK.

Christensen, S., Kristensen, L. M. & Mailund, T. (2001), A sweep-line method for state space exploration, *in* 'TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems', pp. 450–464.

Clarke, E., Grumberg, O. & Peled, D. (2000), *Model Checking*, MIT Press.

Clarke, E. M. & Berezin, S. (1998), 'Model checking: Historical perspective and example (extended abstract)'.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms*, 2nd revised edition edn, The MIT Press.

CPN Tools (2009).
URL: *http://wiki.daimi.au.dk/cpntools/cpntools.wiki*

Evangelista, S. & Pradat-Peyre, J.-F. (2005), Memory efficient state space storage in explicit software model checking, *in* 'SPIN Workshop on Model Checking of Software', pp. 43–57.

Holzmann, G. J. (1997), State compression in spin: Recursive indexing and compression training runs, *in* 'In Proceedings of Third International SPIN Workshop'.

Jensen, K. (1997), *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*, Springer-Verlag.

Rushby, J. (1989), Formal methods and critical systems in the real world, *in* 'Formal Methods for Trustworthy Computer Systems (FM89)', pp. 121–125.

Schmidt, K. (2003), Using petri net invariants in state space construction., *in* 'Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), 9th International Conference', Springer Verlag, pp. 473–488.

Wolper, P., , Wolper, P. & Leroy, D. (1993), Reliable hashing without collision detection, *in* 'In Computer Aided Verification. 5th International Conference', Springer-Verlag, pp. 59–70.