Otto-von-Guericke-Universität Magdeburg

Institut für Simulation und Graphik

Lehrstuhl für Simulation

Pre-processing of Stochastic Petri Nets and an improved Storage Strategy for Proxel Based Simulation

A thesis submitted in partial fulfillment
of the requirements of the degree of

Master of Science in Computer Science

Otto-von-Guericke-Universität Magdeburg

2004

Author: Prasanna Balaprakash

Supervisors: Prof. Dr. Graham Horton
Sanja Lazarova-Molnar

Start of the work: November, 17 2003

End of the work: April, 17 2004

Dedicated to my beloved parents and sisters.....

ABSTRACT

Simulation is a branch of computer science which deals with building the real world entities as models and studying their behavior. Stochastic Petri nets are one such tool for modelling the real world entities. Behavior of models are analyzed by different types of simulation methods. The most common and standard approach is discrete event simulation. But there are still some methods, whose full potential is still to be analyzed. Proxel based simulation is one among them.

This thesis has two goals from the proxel based simulation. First goal is to preprocess the stochastic Petri nets. The motivation behind this goal is to automate the processes needed for proxel based simulation. Second goal is to design an improved storage strategy for proxel based simulation. This goal is motivated by the shorter runtime and lower memory requirements for this simulation approach.

Acknowledgements

I would like to take this opportunity to thank my thesis supervisor Professor Graham Horton, for his guidance, advice and encouragement throughout the course of my studies.

I am very much indebted to my mentor Sanja Lazarova-Molnar for her timely and helpful advices to understand the nature of topic and careful reviews on the algorithms, implementations and the report.

I wish to express, my sincerest gratitude to my father Balaprakash, mother Sakunthala who always supported my studies and gave me everything in life and my love to my beloved sisters Siva Sankari and Deepa Aishwarya

I would also like to thank all of my friends who have made my time in Germany enjoyable. I am thankful to Leo Prakash, Jaggy, Karthik, Lakshmi and Sangeetha for their friendship, encouragement and advice and Peter Celler, Bhavani for spending their valuable time for reviewing my thesis.

Last but not least my friend Arthi who always pray for my success even though she is not with me.

Self-Work Statement

Here with I declare that I have completed this work by myself and only with the help of the stated references.

Prasanna Balaprakash
Matrikelnummer : 166275
Madeburg, 17, April 2004

Contents

Chapter 1

Introduction

1.1 Background

Computers play a vital role in the human society. Their massive computing power is widely used in different applications. The main goal of a computer is, to process large amount of data and convey information. The data may come from any system which we are interested in. Close representation of the real world entities are called as models. Simulation aims at processing the models and conveying information about the behavior of such models over time. From weather forecasting to electronic component failure, simulation has broad range of applications.

The huge processing power of high performance computers can be used to simulate complicated models using mathematical approximations. These simulations can help to reduce the number of expensive real world constructions and experiments to a minimum. They are also very attractive and allow users to change parameters which would not be possible in practice. These simulations can be used to increase efficiency and to save time and resources.

Most of the real world models exhibit randomness in their behavior. This is known as stochastic behaviour. In order to analyze the behaviour of these models over time, we need to introduce the same kind of randomness in the simulation method. The most common approach used for this purpose is discrete event simulation. As a result, the output of this simulation method is also stochastic in nature.

Proxel based simulation is an approach which does not require any randomness during simulation. Even though the model exhibits stochastic behavior, the simulation results do not posses any randomness. The behavior of the model over time, is determined in a complete mathematical and deter-

ministic way. Moreover this approach has another advantage for simulating the models containing rare events. For example, the chance that the Pisa tower will fall in another fifty years is almost zero but not zero. While simulating this model with the discrete event simulation the result will be always zero whereas proxel based simulation approach will produce a result of almost zero. The time and system overhead for simulating fifty years is less for this approach compared to discrete event simulation. The results of proxel based simulation are very accurate. Engineering branches such as reliability and safety, requires a simulation approach of high accuracy. Proxel based simulation approach is suitable for these applications.

1.2 Goals of the Thesis

The current implementation of the proxel based simulation approach needs processed data from the model. Therefore it demands knowledge about the behavior of the model and this simulation approach. It cannot process data from the model but requires a processed data from the model. This information has to be supplied by the user.

We need to start this simulation approach from the models. The models here refers to the Petri nets. This makes the proxel simulation approach free from the knowledge requirements. Therefore we can reduce the usage difficulties of this approach. This motivates us to preprocess the models represented by the stochastic Petri nets. This is our first goal which can be achieved by designing an automatic processing algorithm for the Petri nets. This algorithm provides the necessary information needed for the proxel based simulation.

Our next motivation is to improve the performance of this approach. Accuracy, runtime, memory are some of the factors that determines the performance of this approach. The storage strategy is one of the important factor that affects the runtime and memory needed for this approach. The storage strategy here refers to the data structure. It stores the data and provides the methods for accessing the stored data. The data storage and accessing methods determines the amount of memory and computational time required of this approach. Therefore our next goal is to design a storage strategy with improved runtime and memory requirements. This can be achieved by designing a storage strategy whose data structure can access the data in shorter time and store the data with lower memory usage.

1.3 Organization of the Thesis

This thesis is organized into five chapters. First chapter discussed the background of the thesis. Then it described the goals of the thesis and the motivations. Here it explains the organization of the thesis. The second chapter gives the theoretical introduction into the topic. First part of this chapter describes the fundamentals, which includes some basic information about the Petri nets and its working. Further it discusses the concepts related to the existing approach followed by the concepts required for the proposed approach. The second part of this chapter describes the the existing approach for proxel based simulation. The problems in the existing approach relevant to the thesis topic are described finally in this chapter. Third chapter describes the proposals to solve the problems described in the second chapter. Design and implementation of the proposals are explained in this section. Finally, we describe the contribution of the proposals and define the performance criteria required to analyze the proposed approach. Fourth chapter presents the experimentations and their results based on the performance criteria defined in the chapter three. Each experiment describes the result followed by the explanation. Fifth chapter concludes the thesis with a brief summary followed by some suggestions to improve the proposed approach.

Chapter 2

Fundamentals and Existing approach

2.1 Fundamentals

In this section, first we will introduce stochastic Petri nets as the basic requirement for the entire work which is followed by the state change of the stochastic Petri nets. In this process we extract the reachability graph. This is the starting point of the current implementation. Next we will describe the concepts required for the existing approach. This includes the method of supplementary variables and the proxels based simulation. Finally, we will describe the concepts required for the proposed approach. This includes hashing techniques.

2.1.1 Stochastic Petri nets

Petri nets are a promising tool for describing real world entities as models [Petri nets world 2004]. As a graphical tool, Petri nets can be used as a visual-communication aid similar to flow charts, block diagrams and networks. We are going to use the term *stochastic Petri nets* instead of models for further discussions. Stochastic Petri nets are represented graphically with the following components.

Tokens : Tokens represent the objects of the model. For example, taxis in the taxi stand, passengers in a flight etc are modelled with tokens. They are represented as a *dot*.

Places : Places hold the tokens. These are used to model the locations. The taxi stand is a place which holds the token as taxi. They are represented as a circle.

Transitions : Transition represents the end of an activity. A taxi leaving or coming to the taxi stand, passengers getting off or boarding on the flight are some examples for the activities. There are two types of transitions.

Timed Transitions : These are used to model activities associated with time. To model activities such that, the time required to repair the taxi, boarding of passengers at the specified time etc. They are drawn as a rectangle.

Immediate Transitions : These are used to model activities which does not take time such as activities described by the condition "*immediately*" and "*as soon as*". For example, when a cash automata is free, then the customer can occupy it immediately. They are drawn as a bars.

Input Arcs : Input arcs connects places to transitions. This place is referred as the input place of this transition. These arcs determines the occurrence of an activity. They are associated with multiplicity. Multiplicity is the number of tokens necessary for an activity to occur. After the end of an activity, number of tokens specified as the multiplicity are destroyed from the input place. They are are drawn as arrows.

Output Arcs : Output arcs connects transitions to places. This place is referred as output place. They are also associated with the multiplicity which determines the number of tokens created in the output place, at the end of an activity. They are also drawn as arrows.

Inhibitor Arcs : Inhibitor arcs connects places to transitions. They are also associated with the multiplicity. This inhibits the transition, that is, blocks the activity when the number of tokens present in the corresponding place is greater than or equal to its multiplicity. They are drawn as arrows with circles.

Guard Functions : The transitions may contain guard functions. As the name itself suggests, it will guard the transition. In other words it blocks the activity. This is known as disabling a transition. This function is assigned with checking tokens in the various places. Depending on the checking condition, these functions will disable a transition by returning "*FALSE*".

Initial Marking : Markings represent the discrete states of the system. Due to different transition, the tokens are distributed over various

places of the Petri net. The initial state of the model or the Petri net is known as initial marking.

Consider a production system with two kinds of products (A and B). It manufactures the products according to the client's request. Once the production is over then the system will be ready for next request. Petri net for this production system is shown in the Figure 2.1. It has four places marked as P_0, P_1, P_2, P_3, timed transitions marked as T_1, T_2, T_3, T_4 and an immediate transition as T_5. Initially there is a token in the place P_0. All the input and output arcs have a multiplicity of one and there are no inhibitor arcs or guard functions. Token in the place P_0 represents the state of the system before starting its production. The clients request for the product A is modelled with transition T_1 and B is modelled with the transition T_2. Token in the place P_1 or P_2 represents the state of the system, while manufacturing any one of the product A or B. Production time is modelled with T_3 for the product A and T_4 for the product B. Token in place P_3 represents that state of the system that it has finished the production. As soon as the system finishes the production, it will return to the initial state. Therefore the transition T_5 is an immediate transition.

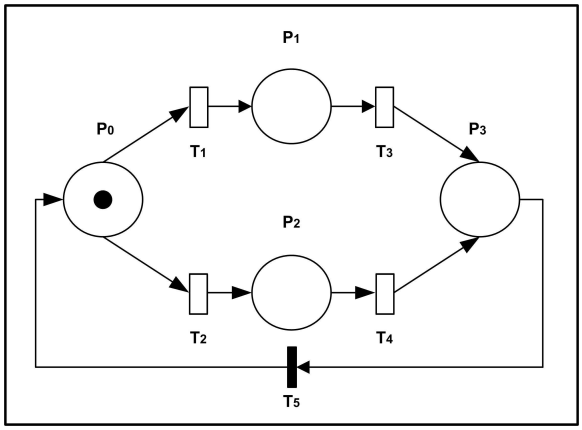

Figure 2.1: Petri net of a simple production unit

Dynamics of the Petri net is exploring the different distribution of tokens, among the places of the Petri nets. Activities creates these dynamics. This is produced as a result of transitions, firing at various times and conditions. Further we are going to describe the reachability graph in this section. In order to understand this concept, we describe the firing conditions which creates the state change in the Petri net. For a transition to fire or it is said to be enabled, the following conditions have to be satisfied:

- The input arc should be active. In other words the number of tokens in the input place of the transition should be greater than or equal to the arc's multiplicity.
- The inhibitor arc should be inactive. That is, the number of tokens in the output place should be less than or equal to the arc's multiplicity.
- The guard function associated with the transition should not return "FALSE".

Satisfying the above conditions makes a transition ready for firing. We will assign the time for the end of an activity. This time is a part of the Petri net specification which describes a transition. This is know as firing time. The firing time assigned to the transition is reduced with time, until it is enabled without interruption. It will fire when reducing time reaches zero. Therefore a timed transitions will take an amount of firing time to fire when it is ready. The immediate transition will fire immediately when it is ready. There are two memory policies of the timed transition:

Enabling memory policy: If a transition is enabled, its firing time is reduced by the time it has been enabled without interruption. These transitions when disabled, have no memory of the amount of time already enabled and must re-sample a new firing time, once they are re-enabled. This is analogous to the ***re-start*** option in a windows operating system, forgetting all the programs which are running before.

Age policy: With this policy a transition that was enabled and becomes disabled without firing, retains its already enabled time and picks up from there without re-sampling, when it becomes enabled again. This is analogous to the ***stand by*** option in a windows operating system, remembering all the programs which are running before.

If a transition fires, then the tokens are re-distributed according to the input arc's and output arc's multiplicity, in the corresponding input and output places. Each such different re-distribution is termed as a marking of the Petri net. We are going to denote the state change as a set of markings. Each marking corresponds to a discrete state of the Petri net. Formally it is denoted by $M = \{m_0, m_1, m_2.., m_n\}$, where n stands for the number of different states that can be reached by a Petri net. Each marking is denoted by $m_i = (P_1, P_2, P_3, .., P_n)$ where each P_k is a number which denotes the number of tokens in the corresponding place. The Figure 2.1 represents the initial state of the Petri net with marking $m_0=(1,0,0,0)$. The state change of the Petri net by the firing rules are as follows:

- The transition T_1 should fire to reach the state with marking m_1=(0,1,0,0).
- The transition T_2 should fire to reach the state with marking m_2=(0,0,1,0).
- From the marking m_1=(0,1,0,0), only possible transition to fire is T_3. This leads to the state change represented by the marking m_3=(0,0,0,1).
- From the marking m_2=(0,0,1,0), only possible transition to fire is T_4. This leads to the state change but to the same marking m_3=(0,0,0,1).
- The transition T_5 is an immediate transition. As soon as the state of the Petri net reaches a state represented by the marking m_3=(0,0,0,1) it goes to the state represented by the initial marking m_0=(1,0,0,0). Therefore the marking m_3 is termed as "***vanishing marking***" and other markings are termed as "***tangible markings***".

We can denote the whole process as a graph with vertex as the marking and the edge as the transition. This graph is known as the "*reachability graph*". This gives the set of all possible discrete states of the Petri net under study. The reachability graph of the Petri net is shown in the Figure 2.2.

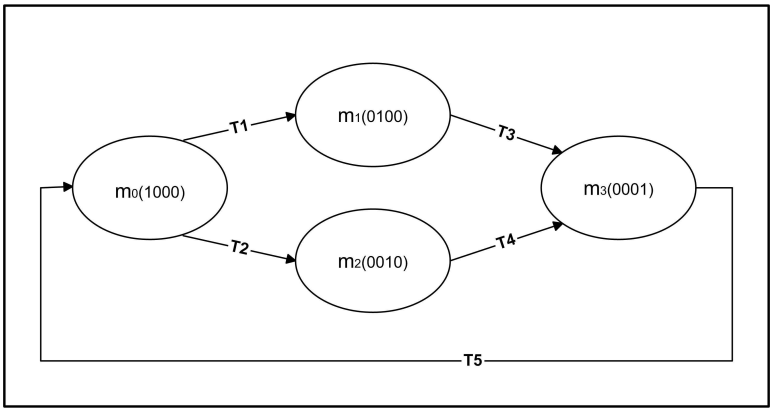

Figure 2.2: Reachability graph of the production unit's Petri net

2.1.2 Method of Supplementary variables

The proxel based simulation is based on the approach of supplementary variables. This approach analyze stochastic behaviour in a deterministic appraoch. This makes the proxel based simulation free from randomness.

We are interested in simulating the models, consisting of both deterministic and stochastic behaviour. In order to model the stochastic nature of the model, the timed transitions can be described with random variables.

Most of the activities occurring in the real world are random. From getting a fever to the break down of our car. No one can say the exact timing for those events. For example, modelling a customers arrival in a bank can take random values because the arrival of the customers is random. It is not possible to say this many customers will arrive in this time. Even though the arrivals take random values, a closer look at these values over time exhibit some characteristics. These characteristics can be described with Uniform distribution, Exponential Distribution, Normal Distribution, Weibull distribution, etc [Random distributions 2004]. Let ϕ be a random variable describes a transition of time duration τ. Then the random firing time of ϕ can be described with

- *Probability Density Function (PDF) represented as* f(τ)
- *Cumulative Distribution Function (CDF) represented as* F(τ)

Consider a transition T described by random variable ϕ of duration τ with its cumulative distribution function F(τ) and corresponding probability density function f(τ). The cumulative distribution measure the probability that a transition fires, with waiting time of τ. There exists another function called survival function described by 1-F(τ). This measures the probability that the corresponding transition has not fired with waiting time of τ. Hence the distribution of a random variable of duration τ can be expressed in three closely related ways:

$$\textbf{Distribution of firing time of T : f}(\tau) \tag{2.1}$$

$$\textbf{Survival Function of T : S}(\tau) = {}_{\tau}\textstyle\int^{\alpha}\textbf{f(u)du} = \textbf{1-F}(\ \tau) \tag{2.2}$$

$$\textbf{Instantaneous Rate Function of T : h}(\tau) = \frac{f(\tau)}{S(\tau)} \tag{2.3}$$

The Instantaneous Rate Function shortly represented as IRF has a reasonably intuitive meaning. It is the probability that a transition having not fired up to time τ will fire during the infinitesimally small interval $\tau+dt$. The Instantaneous rate function determines the number of firings occurring per unit time. Therefore IRF represents the continuous rate of flow of probability for the random variable.

The method of supplementary variables is explained in [German 2000]. First we are going to describe this approach with an analogy followed by the actual approach. Consider two water tanks A and B at a same height. There are two pipes between A and B. One is connected from A to B and another one is connected from B to A. Assume that there is one liter of water in the tank A. As time goes on, the water from A goes to B. Now we can observe the water in B depends on the rate in which the water flows from A. The rate here, refers to two questions,

- How much water is in the tank A ?
- How much time the water is flowing ?

The supplementary variable approach considers the probability as a liquid like water which flows from one state to another state of the model. The main aim of this method is to track the flow of probability mathematically, using the IRF. This IRF is considered to be the rate of flow in the water tank example. The tanks are nothing but markings or discrete states. The questions which determine this probability flow is given by,

- How much probability is in the previous marking ?
- How much time the transition causing the state change is enabled ?

This enables the proxel based simulation to predict the behavior of the model deterministically even though the model is stochastic. The deterministic prediction comes from IRF calculation. The results of this approach are the probabilities of different markings over time. From these results it is possible to predict the behavior of the system because different markings represents different discrete state of the system. The total probability of 1 in the initial marking during the start will be redistributed over time. This redistribution is according to the IRFs of the corresponding transitions which cause changes in the system. The supplementary variables are the measure of a transition 's age intensity τ. In other words, it describes how old a transition is ? The calculation of IRFs is based on this parameter τ. Therefore for an enable memory policy, transitions once disabled is considered as dead and when enabled again it is newly born. But for age memory type. once disabled it is sleeping and when enabled again it remembers the age.

2.1.3 Proxel Based Simulation

Proxel is a new computational unit for analysing the behaviour of models [Horton 2002]. This approach is used for analyzing the behaviour of the stochastic Petri nets. Discrete event simulation is used as a standard approach for this purpose, which requires to simulate the same kind of stochastic nature during the simulation, to predict the behaviour. Therefore the output or result of such analysis will be stochastic in nature. The proxel-based simulation is a deterministic approach based on the method of supplementary variables. It works with the state-space of the model. The idea behind this simulation algorithm is to approximate the discrete stochastic process in a continuous approach, using a discrete time step *dt*. This yields a computational model, consisting of a set of discrete states at each time

step with certain probabilities. The name proxel comes from the analogy of pixel in computer graphics. Pixel is a basic unit consists of the value at x and y co-ordinate in a two dimensional image whereas proxel is data structure with following components:

- marking m_i represents the corresponding discrete state of the system.
- global simulation time t, this reflects the simulation time in steps.
- age intensity Vector.
- p = probability that the model will be in the marking m_i due to the enabling times of the transitions in the age intensity vector.

Consider a model having transitions T_1,..,T_n. Let τ_1,..,τ_n represents the enabling time of each such transitions. Each of them are referred as age intensity. The age intensity vector is a vector which contains an entry for the enabling time of each transition T_1,..,T_n. At time t, the age intensity vector is defined as $\tau(t) = (\tau_1 , ..., \tau_n)$. These variables which constitute the age intensity vector are known as supplementary variables. The value of this variable increases with time, only if the transition is enabled.
It uses age intensities of the transitions as the parameter while calculating the IRF. The proxel as a whole conveys that, the marking m_i occurs with the probability p, at time t, due to the enabling time of the corresponding transitions in the age intensity vector.

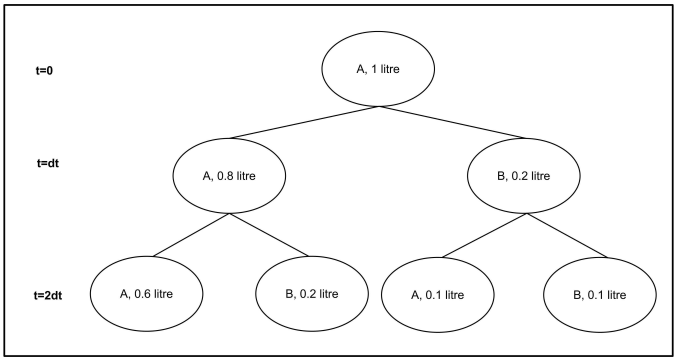

Figure 2.3: Water level over time

Consider the water tank example described in the previous section. There are two pipes such that they will allow uni-directional flow of water between the tanks. If we observe the water over time, let us say 0, dt, 2dt, then the flow of water depends on the time and the amount of water in the flowing tank. Assume we have one litre of water in the tank A. The rate

of flow from tank A to B is 0.2 litre per dt and B to A is 0.1 litre per dt. At each time step the sum of the capacity will be one litre. The level of water in each tank thus depends on two entities. The amount of water and the time for which it is flowing. The water levels in the tanks were described in the Figure 2.3.

We are going to describe the proxel approach with a simple model, based on the analogy explained above. Consider a machine maintenance model with two states, Running and Maintenance. Figure 2.4 shows the Petri net of the model whose reachability graph is shown in the Figure 2.5

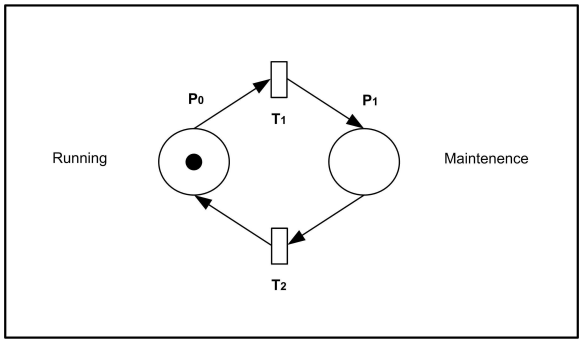

Figure 2.4: Petri net of the maintenance model

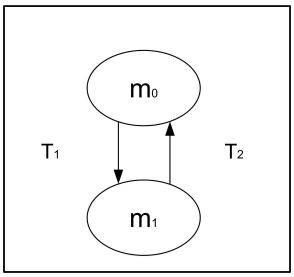

Figure 2.5: Reachability graph of the maintenance model

At time 0, the initial proxel represents the initial marking. It is given a probability of 1. The age intensity vector has an entry for each transition T_1 and T_2 as τ_1 and τ_2. For the initial proxel both of them is zero. This implies that the transitions T_1 is enabled but for no time and transition T_2 is disabled initially. During the next time step *dt*, we have two proxels generated from the initial proxel. One of them represents the state change from marking m_0 to m_1 . Another proxel represents the chance of staying in the same marking. This is similar to the water tank example where we explore the possible water flow from the tanks. The new proxel representing

m_1 has the age intensity vector as (0,0). It tells us that the transition T_1 is fired now. Another proxel representing m_0 has the age intensity vector as (dt,0). This is because the transition T_1 is enabled for the time *dt* but not fired. The second entry is zero because the transition T_2 is not enabled.

Let p_0 and p_1 represents the probability of the proxels, representing the marking m_0 and m_1. They are calculated based on the IRFs and the type of the transition such as Exponential,Uniform etc. This is similar to calculating the water flow from each tanks in the water tank example.

p_1 = 1.0 * h(Enabling time of the corresponding transition i.e T_1)* dt.

p_0 = 1-p_1 .

Proxels tracks the flow of probability from the initial marking of the Petri net to all other possible markings represented in the state space. Therefore the initial probability of one in the initial marking is redistributed to all the possible markings at each time step. The proxel based simulation approach generates a number of proxels for each time step. These proxel represents the different markings of the state space. This approach analyze the model's steady state behavior or the behavior until the specified time. This results in the generation of series of proxels at each time step, from the generated proxels of the previous time step resembling a tree structure. This structure is referred as *proxel tree*. This tree represents the state space in terms of proxels. This is shown in the Figure 2.6.

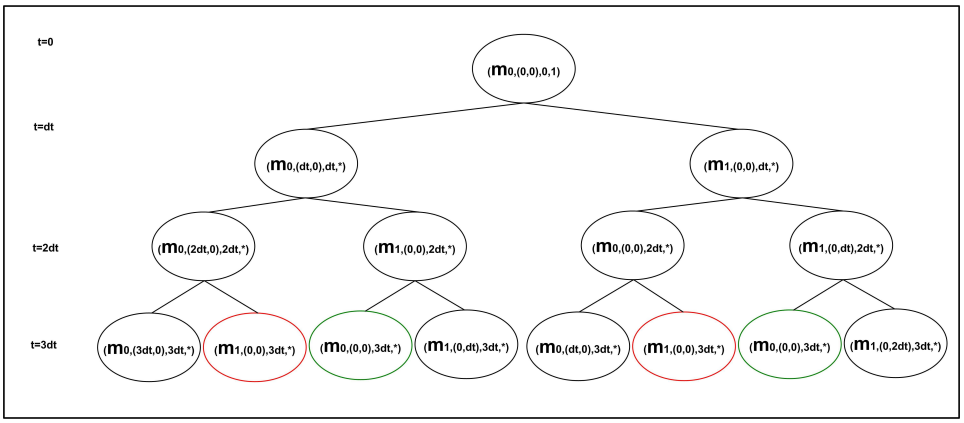

Figure 2.6: Proxel tree of the maintenance model

During the process of generating the proxel tree, at each discrete time step, the same proxels may be generated again and again from different

predecessor. From the Figure 2.6, we can notice that there are two same state proxels containing marking m_0 (green) and another two with marking m_1 (red). The same state proxels refers to two or more proxels with same marking and same age intensity vector. We are not going to store all those proxels. In each case we sum up the probabilities of two proxels in single proxel. This reduces the storage space by having only one proxel of its own kind. If we allow duplicates then we have to process all those proxels in the next time step. Therefore we need to process only a single proxel which reduces the processing time and storage space at each discrete steps.

2.1.4 Hash Functions

Hashing is a searching technique where each search item is characterized by a key, which is a part of the item used to identify it [Introduction to Hash funtions]. In our case, the item refers to the proxel and key refers to the key computed from the proxel. In general a hash function h(k) is represented in the following way,

$$h(k) \quad = \quad (x) \quad mod \quad N \tag{2.4}$$

In this equation the k refers to the item's key and N refers to expected number of items to be stored. The hash function in the hashing technique generates the hash x from which the hash value is computed from Equation 3.1. A hash table is a data structure which consists of two entities. An array with capacity N called as Hash table. The second one is the hash function. Hash function computes a hash from the items key. This hash may be 16, 32 or 64 bit integer. The item is stored in the array by the taking the hash value as the index. If we have an item with its key then we will compute the hash value to know the index where it is stored. Therefore it is possible to find the item in a single access theoretically.

Collisions: In practice it is very hard to design a hash function which will distribute different keys to different hash values. If one or more different keys have a same hash value, then the items are said to be colliding. This is know as "*collision*" in hashing. There are several ways to handle this collision. The simple way of handing the collision is "***linear probing***". There will be another array in the index where the collisions occurs. This is called as "*bin*". All the items which causes collisions in the specified index are stored in the bin. While searching for an item, the key is computed to know the index, where it is stored. Afterwards probing or looking into this bin for the specific item will be done. Performance of this method depends on the load factor α which is defined as :

$$\alpha = \frac{n}{N} \tag{2.5}$$

where n is the number of items stored and N is the dimension of the array where it is stored. Smaller the load factor α better the performance is. The better performance here refers to the condition where the number of collisions is smaller. Another factor which determines the collisions is the length of the key. The length of the key refers to the number of bits in the key. The hashing technique will distribute the keys according to the number of bits in the key. Larger this number, smaller the collisions will be. The worst case for this linear probing for searching an item, is the dimension of the largest bin. This comes from the fact that if we want to search an item which is located at the last index of the largest bin. Searching works well even with poor hash functions, that is the functions that do not distribute the input cleanly. Most automated retrieval mechanisms today are based on hashing. Figure 2.7 shows the hash table with some data items.

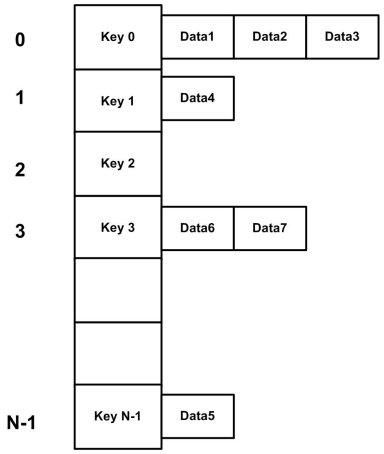

Figure 2.7: Hashing technique

2.2 Existing approach

In this section we describe the implementation details of the existing approach for proxel based simulation. We are going to present the approach with an illustration. The first sub section describes starting point of this approach. Next sub section describes the implementation details of this method followed by some research problems relevant to our thesis topic.

2.2.1 Input Specification

The current implementation has the following details, in the input file used for the proxel based simulation:

- time interval *dt*, known as discreatisation parameter, is the most important parameter which affects the accuracy of this approach. It is specified as a part of the input specification. The question of how close we are approximating ? depends on this parameter. More discussions about this parameter and resultant error is described in [Horton 2002].
- simulation time, the time duration of the simulation, shortly represented as *t*.
- reachability graph of the Petri net.
- transitions with their *type*, *parameter* and *memory policy*.
- age intensity vector containing a supplementary variable τ_i for each the transition T_i of the Petri net.
- initial marking m_0 which represents the initial state of the Petri net.

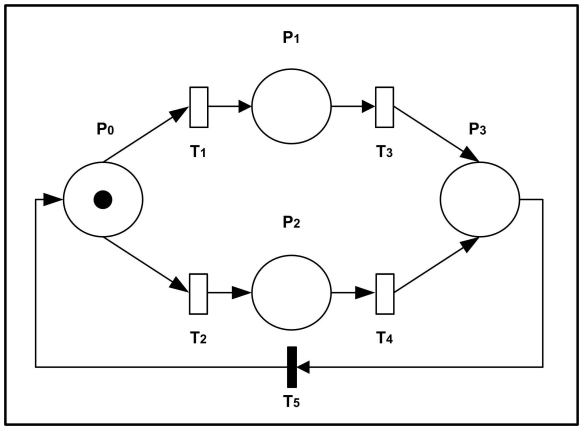

Figure 2.8: Petri net model for the illustration

In the Petri net shown in the Figure 2.8 assume the transitions T_1 and T_2 belongs to the age memory policy. T_3 and T_4 are of enable memory type. The input specification of the current implementation consists of the following details.

- time interval *dt*, 0.2
- simulation time *t*, 50.
- reachability graph can be represented in several ways. One such way is a reachability matrix.The row and column corresponds the the marking and each entry in the matrix corresponds to the transition. The

entry in i,j of the matrix is transition then that transition will make the change from marking m_i to m_j. We define ø for no such transition from those markings. The matrix for our study model is shown below,

Marking	0	1	2	3
0	ø	T_1	T_2	ø
1	ø	ø	ø	T_3
2	ø	ø	ø	T_4
3	T_5	ø	ø	ø

- age intensity vector

$$\overrightarrow{\tau} = (\tau_1 \ , \tau_2 \ , \tau_3, \tau_4)$$

Let τ_1 , τ_2 , τ_3 and τ_4 represent the elapsed time of the transitions T_1, T_2, T_3, T_4 respectively in our study model. The immediate transition T_5 does not require such variable because there is no time delay for firing T_5 when it is enabled.

2.2.2 Implementation Details

The implementation of the current approach works on the state space of the Petri net. The state space of the example Petri net is shown in the Figure 2.9. We describe the steps in the current implementation for our example.

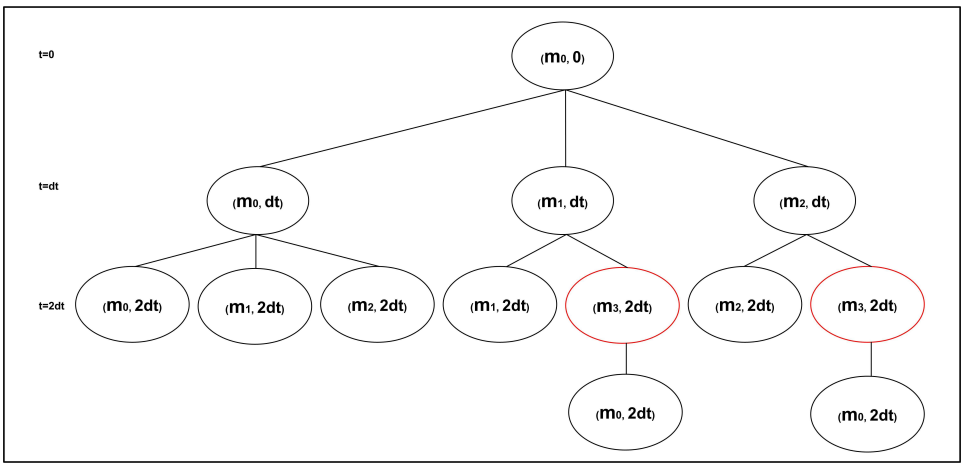

Figure 2.9: State space of the Petri net model

Initialization: We denote the initial proxel as P_0. This is required to start the simulation algorithm.

1. The marking number of P_0 is assigned to 0 representing the initial marking m_0.
2. The age intensity vector of P_0 is represented as $\overrightarrow{\tau}(0) = (0, 0, 0, 0)$.
3. The probability p_r is assigned as one. Because the only possible discrete state is represented by this proxel with the marking m_0.

Storage strategy : For storing the proxels, unbalanced binary tree is used as the data structure [Lazarova-Molnar and Horton 2003 A]. Two such trees are used for the current implementation denoted as BT*current* and BT*next*. The tree BT*current* contains proxels for processing during the current time step. While processing these proxels new proxels are created which are stored in BT*next*. It contains all the proxels to be processed for next time step. After processing BT*current* then this tree no longer need. Therefore BT*current* is deleted and the tree BT*next* is assigned to BT*current*, leaving BT*next* empty. This process is repeated for each time step. The Figure 2.10 shows the entire process of proxel generation. This represents the state space of the Petri net in terms of proxels.

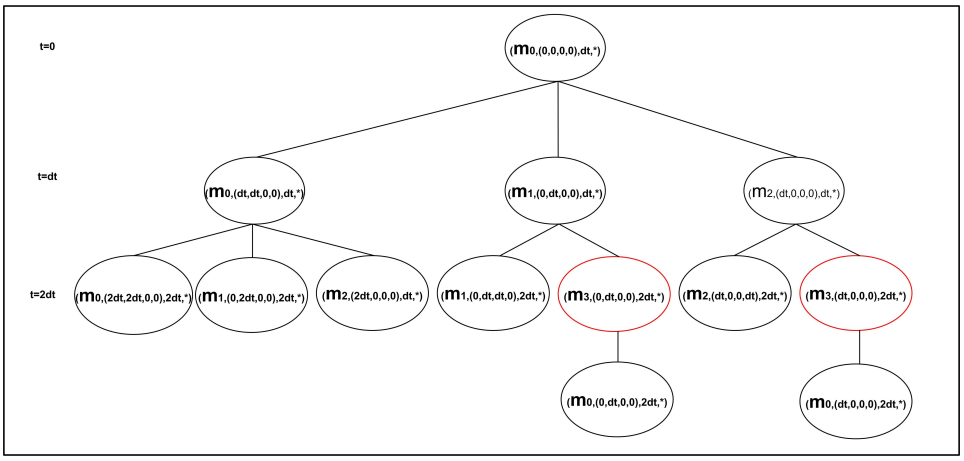

Figure 2.10: Proxel tree of the study model

The tree BT*current* is initialized with the initial proxel. The key used to store the proxel is calculated from the age intensity vector and marking of the corresponding proxel.

Generation of proxels : In the next time step, new proxels are generated for each transition in $\overrightarrow{\tau}$ which are enabled in the marking m_0. For our study model there are two such proxels due to the enabled transitions T_1 and T_2 from the marking m_0. These two proxels with markings m_1 and m_2 represents the condition that the transitions T_1 and T_2 are fired respectively. We also have another proxel with same marking m_0 which models the condition, if both of the transitions are enabled but does not fire.

Changes in proxel's components : After a new proxel is generated in the above step then its proxel components should be updated.

1. marking number of the new proxel should represent the new discrete state,
2. global simulation time is incremented with time step *dt*,
3. The probability of this proxel is calculated from the IRF of the age intensity τ_i of the transition T_i,
4. Updating $\overrightarrow{\tau}$ has the following steps.

   If a proxel represents the transition(s) enabling condition and staying in the same marking, then its entry(s) τ_i in $\overrightarrow{\tau}$ is incremented with *dt*.

   If a proxel represents the condition in which the transition T_i is fired, then its entry τ_i in $\overrightarrow{\tau}$ is made zero. The other transitions enabling time, which are disabled by this transition is updated as follows,

   If the transition T_j memory type is *age*, then its corresponding entry τ_j is kept as it is.

   If the transition T_j memory type is *enable*, then its corresponding entry τ_j is made zero.

The age intensity vector $\overrightarrow{\tau}$ of the three proxels generated in time step *dt*, from the initial proxel time step at time step 0, is updated according to the above conditions. For the same state proxel which represents the marking m_0 is given by, $\overrightarrow{\tau}$ = (dt , dt, 0, 0). It models the condition that the transitions T_1 and T_2 were enabled for time *dt* without firing. For the proxel containing m_1 is given by, $\overrightarrow{\tau}$ = (0 , *dt*, 0, 0). The 0 for τ_1 comes from the fact that T_1 is fired and *dt* for τ_2 from the age policy of T_2, such that it is enabled for time *dt* and

disabled now. Similarly for the proxel containing m_2 is given by, $\overrightarrow{\tau} = (dt\ ,\ 0,\ 0,\ 0)$

We assign the probability to the proxel containing the marking m_0 after calculating the IRFs of the other proxels. This comes from the fact that the remaining probability after calculating the IRFs of the other proxels, is the probability to remain in the same marking.

Storage of proxels : After the generation of each proxel the proxels are stored in the tree BT*next*. For such proxel, the key is computed form the marking and age intensity vector. Key comparison is made with the existing proxels, to store the new proxel in BT*next*. Figure 2.11 shows the binary tree with proxels at time step *dt*.

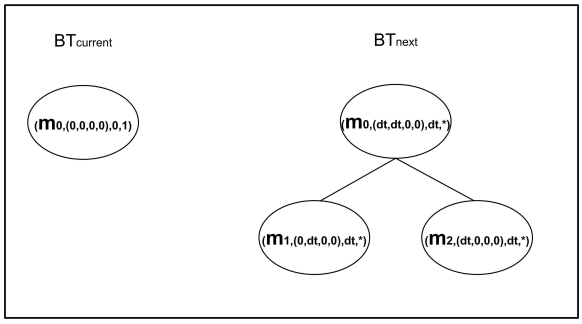

Figure 2.11: Binary tree at time step *dt*

Next we describe the role of storage strategy employed here. The proxels representing the same state with same age intensity vector are combined by adding their probabilities into a single proxel. When a new proxel is generated we search for the occurrence of same proxel in BT*next*. If we find such proxel then we will add the new proxel's probability to that proxel. Otherwise we add the proxel to the tree. Here it takes logarithmic time to search a proxel. The best case for searching will be O(log N) where N is the number of proxels in the searching tree BT*next*. This requires every proxel in the tree to have a right and left child proxel.

Simulation: The results of this implementation are the probabilities of the markings in the steady state or over specified time. Therefore each proxel's probability is stored in a separate array whose index corresponds to the marking's number. For example in the current step, we store the proxel's probability containing marking m_0 to the array whose index is 0. If we have another proxel with same marking m_0

then irrespective of age intensity, the probability is added to the same index 0. This array is generated for each time step.

Now we remove BT*current*. BT*next* act as BT*current* for the next time step. We repeat the steps explained in "*Generation of proxels*" to "*Simulation*" at each time step. While moving to the next time step, global simulation time t of each proxel is incremented by *dt*. We stop the simulation when t reaches the specified simulation time. Figure 2.12 shows these changes in the binary tree with proxels at time step *2dt*.

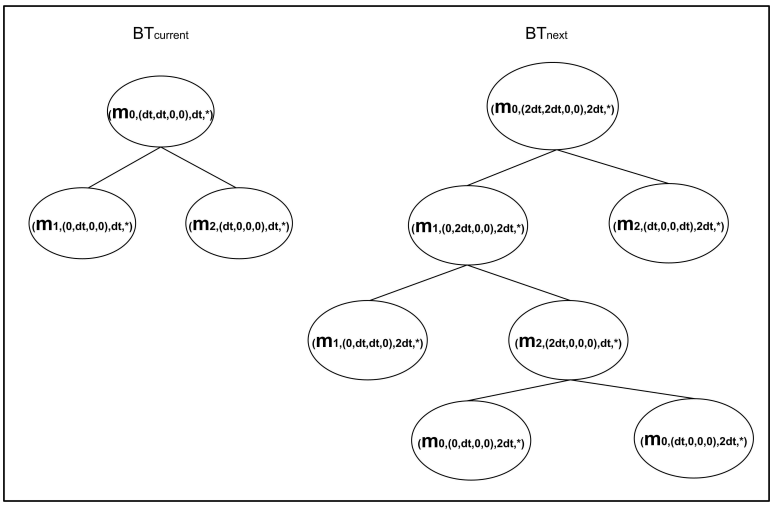

Figure 2.12: Binary tree at time step *2dt*

A note on vanishing marking: Markings in which immediate transitions can be enabled are termed as *vanishing markings*. There is no need to store the proxels containing vanishing marking. Instead the successor proxel is found from these proxels without incrementing *dt*. That is the system will go to next marking without time delay. The probability and age intensity vector associated with these proxel are assigned to the successor proxel. [Lazarova-Molnar and Horton 2003 C] explains this approach.

A note on probability value: The current implementation has threshold value for probability. The threshold value is used to discard the proxels. If the probability value of the proxel is less than the threshold value for example 10E-12 then it is discarded [Horton 2002]. This comes from the fact that the probability of occurrence of those markings with the given age intensity vector is very very less. They do not

have a significant effect in the overall probability. This aids in the storage space and runtime improvement significantly.

2.2.3 Open Questions

This section describes some of the research problems raised by this implementation, relevant to our thesis topic. These problems are the motivation behind the proposed approach.

Advantages of preprocessing

The existing implementation for proxel based simulation is based on the reachability graph. Therefore the current implementation requires the knowledge about the dynamics of the Petri net and proxel approach. The motivation of constructing an automated interface which processes the Petri net prior to the simulation forms our first goal. Therefore the Petri net is processed to obtain the reachability graph. We propose to present the reachability graph in the form of reachability matrix.

The age intensity vector $\overrightarrow{\tau}$ has entries for each of the transition T_1 ,.., T_n as τ_1 ,.., τ_n. It is observed in most of the cases, that all the entries in $\overrightarrow{\tau}$ are not relevant for each marking of the Petri net. For example consider our study model. The $\overrightarrow{\tau}$ in marking m_0 have to include only the enabling times of transitions T_1 and T_2. Because only these two transitions are enabled in the marking m_0. Hence it is enough to store τ_1 and τ_2 in $\overrightarrow{\tau}$ for marking m_0. Similarly for other markings such as m_1 and m_2 needs τ_3 and τ_4 respectively. Now it is possible to define a marking dependent age intensity vector. Based on this argument we have the following age intensity vector for each marking.

Marking	Variables
m_0	τ_1 , τ_2
m_1	τ_3
m_2	τ_4
m_3	∅

Proxel simulation approach needs to remember the age intensities of the transitions T_1 and T_2 when they are disabled. Because both of them belongs to age memory policy. When transition T_1 fires from marking m_0 then the transition T_2 is disabled. This leads to the argument that the enabled time τ_2 of transition T_2 while coming to the same marking m_0 again has to be remembered. This happens, when we reach the marking m_0 via m_2 and m_3. Therefore these markings has to remember τ_2. The similar argument holds for τ_1 for the markings m_1 and m_3. Therefore the final marking dependent

age intensity vector is specified as:

Marking	Variables
m_0	τ_1,τ_2
m_1	τ_3,τ_2
m_2	τ_1,τ_4
m_3	τ_1,τ_2

We describe the above process as finding the optimal supplementary variables for each marking [Horton 2002]. This aims at some kind of improvement for the proxel simulation approach. The intuitive explanation for optimization can be explained as follows. First, the reduction of key computation time. While storing, key for a proxel is computed from age intensity vector and marking. Larger the age intensity vector, larger will be the proxel key. This increases computational time for the key computation and updating the age intensity vector $\overrightarrow{\tau}_i$. The second part, memory required for each proxel. By having the less entries for $\overrightarrow{\tau}_i$ for each marking m_i, the amount of memory required for each proxel is reduced by 32 bits (implementation specific) for an irrelevant entry.

Storage Strategy

We described about the storage strategy employed in the existing approach. Unbalanced binary tree is used for storing proxels. In best cases, the time required to search a proxel is O(log N). This requires the tree to be balanced. In other words, each proxel in the tree should have a right and left child proxel. We need to do some balancing when the tree does not satisfy the condition. The time required for this affects the simulation runtime. The time required for balancing the tree is more and the tree without balancing performs well than balanced tree. Therefore it was decided to use the unbalanced binary tree [Lazarova-Molnar and Horton 2003 B] without any balancing.

In worst cases, the time required to search a proxel in this tree will be O(N). It may take 1000 access to search a proxel in a tree containing 1000 proxels. This will happen when the proxel is added to the tree either as a left or right child always [Algorithms and Data Structures]. Further if the simulation is done for a very long time or for complex models then the number of combining proxels will be more. If the time required to search a proxel during these conditions is linear, then the simulation time increases to a large extent. Therefore we need an improved storage strategy to overcome this problem. Introduction of this problem and a study is made in [Lazarova-Molnar and Horton 2003 B]. This goal is a supplementary work to that research. This forms the motivation for our next goal of designing an

improved storage strategy for proxel based simulation This aims at reducing the searching time and storing time of the proxels. The improvement here refers to the improved runtime and memory requirements.

Chapter 3

Proposals for improving the existing approach

In this section we describe our proposals for improving the existing approach. We explain our proposals in two steps. The first part explains the method employed for solving the problems related to the preprocessing of Petri nets. Next part describes the storage strategy which can be used for proxel based simulation.

3.1 Pre-processing of the Petri nets

The preprocessing is an automated process to reduce the usage complexity of the current implementation. We propose this part has to be included with proxel simulator. Therefore the proxel based simulator requires only the Petri net specification. We divide the preprocessing process into the following steps,

- Specifications of the input Petri net file.
- Generating the reachability graph from the Petri net.
- Defining marking dependent age intensity vector.

3.1.1 Specifications of the Petri net

In this section, we describe the ways of specifying a Petri net file. The preprocessing approach works on this specification. Intuitively the specification defines the Petri net. We number the transitions as T_1,..,T_n, places as P_0,..,P_n. We use the Petri net shown in the Figure 3.1 for the explanations. The components of the input Petri net specification file are as follows.

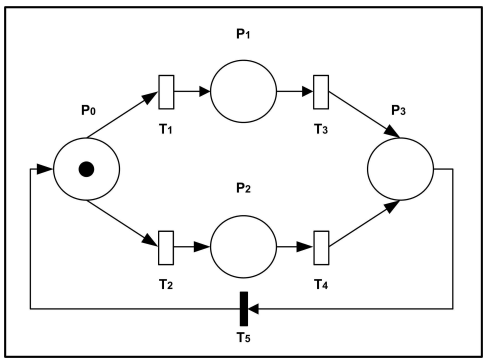

Figure 3.1: Petri net of the study model

Number of components: The number of places, transitions, input arcs, output arcs and inhibitor arcs in the Petri net are specified in this part of the file.

Arcs properties: In this component we specify the connecting places and transitions of the arcs along with its multiplicity. Each type of an arc has its own specification. They are as follows:

Input arc: (Place Transition Multiplicity)

Output arc: (Transition Place Multiplicity)

Inhibitor arc: (Place Transition Multiplicity)

Initial marking: The initial state of the Petri net is specified in this component as the number of tokens in each place as follows:

(# P_0,# P_1..# P_n)

Transitions properties: Each transition in the Petri net is specified as a 4 tuple entity.

(T_n,Type,Parameters,Policy)

where n in T_n is the number obtained during the numbering process. The next entry is the type, if it is timed then it is specified as E for Exponential, D for Deterministic, U for Uniform, etc., or if it is immediate then it is specified as I. Followed by type is the parameters. For immediate transition the parameter will be the probability. For other transitions, it depends on the data from the model. The last entry is the type of the policy. We specify "***A***" for age memory policy and "***E***" for enable memory policy.

Simulation time: This is the time $\boldsymbol{t}$ up to which the simulation has to run.

Time step: This is the discretisation parameter dt as discussed in the previous chapter.

Now we specify the example Petri net shown in the Figure 3.1 as follows.

Specification of the petri net
Number of Places Transitions Input-arcs Output-arcs Inhibitors
4 5 5 5 0
Input arcs (Place → Transition Multiplicity)
P_0 T_1 1
P_0 T_2 1
P_1 T_3 1
P_2 T_4 1
P_3 T_5 1
Output arcs(Transition → Place Multiplicity)
T_1 P_1 1
T_2 P_2 1
T_3 P_3 1
T_4 P_3 1
T_5 P_0 1
Inhibitor arcs(Place → Transition Multiplicity)
Initial marking
1 0 0 0
Transitions (Number Type Parameter Policy)
T_1 U 1.5 3.3 A
T_2 U 1.5 3.3 A
T_3 U 1.5 3.3 E
T_4 U 1.5 3.3 E
T_5 I 0.3
Maximum Time
50
Time step
0.2
End of the specification of the Petri net

3.1.2 Reachability graph generation

In this section, we explain the approach to generate the reachability graph from the Petri net specification. For this purpose, we use popular breadth-first-search algorithm in computer science. The firing rules of the Petri net are added to this algorithm[Gianfranco ciardo and Andrew S. Miner 2002]. The breadth-first-search technique explore all the possible discrete states created by the transitions in the Petri net. The implementation of the algorithm assumes the Petri net is bounded. Boundedness here refers to the set of all markings generated by the Petri net is finite. We present here the reachability graph of the Petri net in the Figure 3.2.

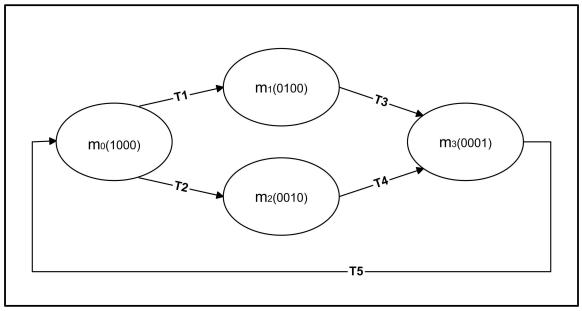

Figure 3.2: Reachability graph of the Petri net under study

Algorithm Reachability

completedMarking = ϕ ;
foundMarking = { Initial marking }

while *foundMarking* $\neq \phi$
do begin
    i = RemoveMarking(*foundMarking*)
    completedMarking=*completedMarking* $\cup$ i
    for each transition T that is enabled in i
    do begin
        j = GenerateNewMarking(i,t);
        SearchInsert(j, *completedMarking*$\cup$*foundMarking*, *foundMarking*)
    end;
end;

The description of the algorithm is as follows. This algorithm starts with the initial marking specified in the input Petri net file. The algorithm uses two data structures, *completedMarking*, used to keep track of the generated markings which forms the reachability matrix and *foundMarking* which is used to store the markings temporarily. The algorithm stores the generated markings in both of these structures. The markings in *foundMarking* is removed while generating the successor markings from it. These successor markings are generated by the function **GenerateNewMarking**. This function generates new markings j with the enabled transitions in the marking i. The function **SearchInsert** search for the occurance of the new marking, in *completedMarking*. If the marking is already generated then it wont add the marking, otherwise it adds the new marking to *completedMarking* and *foundMarking*. When the algorithm explores the possible markings of the Petri net then *foundMarking* becomes empty. This make the algorithm to stop further processing. Finally this algorithm produces a reachability matrix which contains the markings of the Petri net as row, column index and transitions as the element of the matrix.

We illustrate this algorithm to our Petri net model. Initially the marking m_0 is stored in *foundMarking*. Now the algorithm start exploring the possible markings. It removes the initial marking from *foundMarking* and store that marking in *completedMarking*. Then it finds the successor marking with **GenerateNewMarking**. This results in two markings m_1 and m_2 due to the enabled transitions T_1 and T_2 from marking m_0. These two markings are stored in *foundMarking* and *completedMarking*. Next step, the algorithm removes marking m_1 from *foundMarking* and generate new marking m_3 due to transition T_3. Again this is stored in *foundMarking* and *completedMarking*. Then it removes the marking m_2 from *foundMarking* and generate the new marking m_3 due to transition T_4. This marking is already a generated marking, so the function SearchInsert does not allow this marking to be added in *foundMarking*. Finally the marking m_3 is processed to get m_1 due to the transition T_5. This is also a marking generated in previous steps of the algorithm. So *foundMarking* will become empty to stop this algorithm. The algorithm performs a breadth-first-search for our study model Petri net and produce a reachability matrix as mentioned below,

Marking	m_0	m_1	m_2	m_3
m_0	ø	T_1	T_2	ø
m_1	ø	ø	ø	T_3
m_2	ø	ø	ø	T_4
m_3	T_5	ø	ø	ø

This matrix not only have the possible markings of the Petri net but also the information about the tangible and vanishing markings. The vanishing

marking is one which has an entry of an immediate transition. In our case it is marking m_3 which has an entry of T_5 termed as vanishing marking.

3.1.3 Marking dependent age intensity vector

This section describes our design and approaches in finding age intensity vector for each marking. The age intensity vector has entries which are relevant to the marking. In this process, we reduce the constant size age intensity vector of each marking to variable sized vector. The variable size has only the entries for transitions which are relevant for the marking. The question of relevancy has a broad meaning of optimality. Therefore we restrict our relevancy to the guidelines which we designed for this purpose. Hence we refer to this implementation as an algorithm to find a *nearly optimal number of supplementary variables.*

Guidelines for the Optimality

In order to find a nearly optimal number of supplementary variables for each marking, we describe a set of guidelines designed according to the study and analysis of the Petri nets. Each of these guidelines is described with a scenario using our reachability graph as follows:

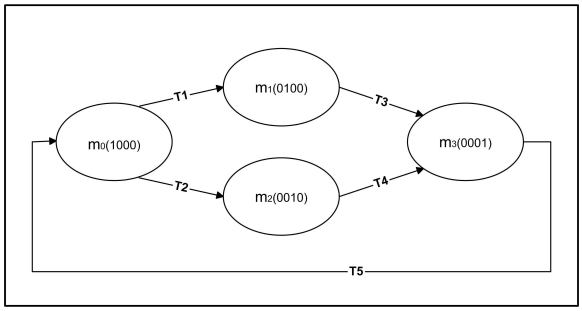

Figure 3.3: Reachability graph of the Petri net under study

1. If the marking has one or more transitions, then it has to remember the enabling time of the transition, until firing. We denote the age intensity vector for each marking as

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_3
$\overrightarrow{\tau}_2$	τ_4
$\overrightarrow{\tau}_3$	τ_5

2. A vanishing marking does not need to remember τ_i, for the immediate transitions T_i enabled in that marking. This comes from the fact that immediate transition fires without time delay. In our case marking m_3 is a vanishing marking hence our age intensity vector reduces to ,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_3
$\overrightarrow{\tau}_2$	τ_4
$\overrightarrow{\tau}_3$	ϕ

3. In the Figure 3.3, from the marking m_0, transitions T_1 and T_2 are enabled at the same time. Suppose we have the memory policy of transitions T_1 and T_2 as ***enable***, then the marking m_0 needs only one variable for storing the enabling time of both the transitions. This comes from the argument that if one of the transitions is fired, then the disabled transition does not need to remember the time for which it was enabled. In this case we specify the marking dependent age intensity vector as,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ
$\overrightarrow{\tau}_1$	τ_3
$\overrightarrow{\tau}_2$	τ_4
$\overrightarrow{\tau}_3$	ϕ

4. Consider the case in which the transitions T_1 and T_2 has ***age*** memory policy. Then the marking m_0 has to remember the enabling times of both the transitions. Because while coming to the same marking m_0, the IRF's of successor markings were based on the enabled time of the disabled transitions. This leads to the following specification,

$$\overrightarrow{\tau}_0 = (\tau_1, \tau_2)$$

5. Consider the same scenario as before. We have to remember the enabled time τ of the disabled transition . Therefore the successor markings has to copy this τ. While coming to same marking again this value is copied to the respective τ in the marking m_0. The transition T_1 has the following effect on its successor markings,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_1
$\overrightarrow{\tau}_2$	
$\overrightarrow{\tau}_3$	τ_1

Similarly the transition T_2 has the following effect on its successor markings,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	
$\overrightarrow{\tau}_2$	τ_2
$\overrightarrow{\tau}_3$	τ_2

Due to the effect of transitions T_1 and T_2, intensity vector for each marking is specified as the union of these vectors as mentioned below.

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_1
$\overrightarrow{\tau}_2$	τ_2
$\overrightarrow{\tau}_3$	τ_1,τ_2

6. During the above process of adding τ_i to the successor markings, if transition T_i is enabled in any of the successor markings, then it will be fired there. Therefore we need not remember the τ_i from the marking, where it is enabled again, to all other successor markings.

7. Let us assume the transitions T_1 and T_2 have ***age*** and ***enable*** memory policies respectively. The marking m_0 has to remember the enabling time of transition T_1 and T_2 until both are fired. If T_2 fires then the marking m_0 has to remember the enabling time of T_1. Effect of remembering τ_1 to successor markings is explained in case [5]. But the marking m_0 need to remember the τ_2 of T_2, only for the time, it is enabled. Therefore there will not be any effect of adding τ_2 to its successor markings. Due to the effect of transitions T_1 and T_2 intensity vector for each marking is specified as the union of these vectors as mentioned below.

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_1
$\overrightarrow{\tau}_2$	
$\overrightarrow{\tau}_3$	τ_1

8. If a transition is specified as Exponential type of memory policy ***age***, then it is considered as Exponential of memory policy ***enable*** . This comes from the fact that, exponential distribution is always memoryless.

9. Consider the same example Petri net without the transition T_5. Then we don't need to remember any of the disabled transitions T_1 and T_2 though they have ***age*** memory policies. Also in this case we need a single τ for the marking m_0. This comes from the argument in this case, that a transition once disabled will not be enabled in the future. Therefore we need not remember the enabled time even though the memory type is age. This makes our making dependent age intensity vector as,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ
$\overrightarrow{\tau}_1$	τ_3
$\overrightarrow{\tau}_2$	τ_4
$\overrightarrow{\tau}_3$	$\emptyset$

10. No duplicate of supplementary variable is necessary for a marking dependent age intensity vector. Different markings can have same disabled transitions. While adding those τ_i in the successor markings, if we have the τ_i already in that marking, then we do not need to add this τ_i again.

Algorithm for finding *nearly* optimal number of supplementary variables

With the set of described guidelines, we designed an algorithm that will compute a *nearly optimal number supplementary variables* for each marking. Each such variables are stored as marking dependent age intensity vector. The specification of the algorithm is followed by the description.

Algorithm Supplementary

Line 01: begin
Line 02: $\forall$ tangible m_i ε $\Re$
Line 03: do begin
Line 04: T_i = getTransitions(m_i);
Line 05: if ($\forall$ T.MemoryPolicy in T_i = "ENABLE")
Line 06: addTransitions(m_i,V_i,T_{i1});
Line 07: else
Line 08: addTransitions(m_i,τ_i, $\forall$ T in T_i);
Line 09: end;
Line 10: $\forall$ (τ_i) $\in$ m_i where # (τ_i) > 1
Line 11: do begin
Line 12: $\forall$ T_i $\in$ (τ_i)
Line 13: {nextMarking} = $\emptyset$;
Line 14: {processedMarking} = $\emptyset$;
Line 15: m_{next}=succ(m_i,T_i) ;
Line 16: {nextMarking} = {nextMarking} $\cup$ m_{next} ;
Line 17: S = τ_i-T_i;
Line 18: {processedMarking} = {processedMarking} $\cup$ m_{next};
Line 19: while(#(nextMarking) > 0 && # (S) > 0)
Line 20: do begin
Line 21: m_{next}=removeFirst{nextMarking};
Line 22: S = {S} - ($\forall$ T_{next} $\in$ (τ_{next}));
Line 23: $\tau_{i,next}$ = {S};
Line 24: $\forall$ T_{next} $\in$ ($\tau_{i,next}$);
Line 25: if (succ(m_{next},T_{next}) $\cap$ {processedMarking} = $\emptyset$)
Line 26: do begin
Line 27: {nextMarking} = {nextMarking} $\cup$ succ(m_{next},T_{next});
Line 28: {processedMarking} = {processedMarking} $\cup$ succ(m_{next},T_{next});
Line 29: end;
Line 30: end;
Line 31: if # (S) = 0
Line 32: $\forall$ T_{next} $\in$ $\tau_{i,next}$
Line 33: (τ_i)$\cup$ $\tau_{i,next}$
Line 34: end;
Line 35: return τ_i $\forall$ m_i;
Line 36: end;

Line 04 to 09 The algorithm computes the supplementary variables only for the tangible markings in the reachability graph $\Re$. It leaves out the computation for the vanishing markings, satisfying the guideline No:2. It gets all the transitions enabled in each marking m_i by the function getTransition(m_i). Next, it checks the each transitions memory policy, for each marking. If all the transitions enabled in the marking m_i have enable memory policy, then it adds only one variable from the guideline No:3. Otherwise it add one variable for each transition to the age intensity vector τ_i for the marking m_i, satisfying the guideline No:1.

Line 10 It process the marking with τ_i computed from Line 04 to 09, whose size is greater than one. Because there are more than one transition enabled in the marking m_i.

Line 12 For each such enabled transitions of the marking m_i, the algorithm computes the supplementary variable as follows.

Line 13 and 14 It initializes two empty sets *nextMarking* and *processed-Marking*. These sets, keeps track of the markings to be processed and markings which already processed for each of the transition in process.

Line 15 The function succ(m_i,T_i) returns the successor marking from m_i for the enabled transition T_i.

Line 16 This successor marking m_{next} is added to the *nextmarking*, so that it can be processed for the successive steps.

Line 17 Let S be a set of transitions which are disabled by the currently enabled transition T_i.

Line 18 Next marking m_{next} is added to the *processedMarking*. The algorithm avoids to process the same marking if it is generated in the successive steps.

Line 19 The condition (S) > 0 implies that, the supplementary variables of the disabled transitions, are added to successive marking, until it is fired in the successive markings. The condition checking *nextmarking* as empty implies that there are still some successive markings from the current marking. When this becomes false with other condition being true then we never reach the marking where we have started.

Line 21 The function removeFirst{nextMarking} removes the first entry in the set for processing. This is termed as the m_{next}.

Line 22 All the τ from the age intensity vector of this marking is removed from S. If the marking m_{next} has transitions which which belongs to S, then it is enabled in this marking and to generate the successive marking. Therefore the disabled transition is enabled in marking m_i.

Line 23 The vector $\tau_{i,next}$ holds the supplementary variable for the marking m_{next}. By guideline [5]

Line 24 to 30 For each transition enabled from the marking m_{next} we have successive markings. We add those reachable markings in both the *nextMarking* and the *processedMarking*, if it is not generated already. Therefore the processed markings can be tracked from *processedMarking*. By checking this set, the algorithm add the marking to both of the sets.

Line 31 to 34 If S is empty, then the marking from where the transitions are disabled can be reached and enabled again, or the transition is enabled in the in any of the successive markings while returning to the marking where it is disabled. For each disabled transition the algorithm will compute age intensity vector for successive marking. Due to the fired transition in m_i, the age intensities of the disabled transitions has to be remembered by the markings in the successive path. This is given by $\tau_{i,next}$. In other words due to the transition T_i from marking m_i the markings m_{next} has to remember the age intensities of the disabled transition. These vectors are combined finally to get the final marking dependent age intensity vector for each marking.

Line 35 Finally the algorithm returns the age intensity vector τ_i, for each marking m_i.

We illustrate this algorithm with our study model Petri net. The first part of the algorithm, line 04 to 09, produces the following result.

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_3
$\overrightarrow{\tau}_2$	τ_4
$\overrightarrow{\tau}_3$	ϕ

Now it will process only the age intensity vector $\overrightarrow{\tau}_0$. This vector has the capacity greater than one. The algorithm computes age intensity vector for each marking, line 10 to 28, modelling the possibility of the disabled transitions T_1, T_2. With T_1 as disabled condition, the second part computes age intensity vector for the successive markings. This will result in the following vector for each marking.

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	
$\overrightarrow{\tau}_1$	τ_1
$\overrightarrow{\tau}_2$	
$\overrightarrow{\tau}_3$	τ_1

For the second iteration, it takes T_2 as the disabled transition and produce age intensity vector for the successive marking as follows,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	
$\overrightarrow{\tau}_1$	
$\overrightarrow{\tau}_2$	τ_2
$\overrightarrow{\tau}_3$	τ_2

The final part of the algorithm,Line 31 to 34, combines all these age intensity vector and produces a single age intensity vector for each marking. This is shown below,

Age intensity vector	Variables
$\overrightarrow{\tau}_0$	τ_1,τ_2
$\overrightarrow{\tau}_1$	τ_3,τ_1
$\overrightarrow{\tau}_2$	τ_4,τ_2
$\overrightarrow{\tau}_3$	τ_1,τ_2

3.2 Storage strategy

In this section, we describe our second proposal of designing an improved storage strategy for proxel based simulation. We describe the potential problems in some of the standard data structures followed by the design of the proposed approach.

3.2.1 Potential problems in standard data structures

We selected some of the data structures which effectively supports two operations, searching and storing. These are the operations required for the implementation of the proxel based simulation. These operations should be quick enough for storing and searching a proxel in the data structure.

Array

In this case, we use a single dimensional array of capacity *N*. The range of the index of the array is [0..N-1]. The key is computed for the given proxel in the specified range. The probabilities are added if there is a proxel in the computed index while searching for a proxel in the array. Otherwise the proxel is placed in the computed index. Therefore given a proxel key,

searching the proxel takes single access O(1) in the data structure.

The main disadvantage of this approach is designing an array, such that it can hold all the possible values produced as the key of the proxels. Further the key should be unique. This requires very large value of N and expensive key computation. Therefore the array size will become very large and key computation is expensive in terms of running time.

Dynamic array

This data structure aims to avoid the very large value of N, as discussed in the previous section. Instead of having fixed N for the array, the capacity of the array is increased, when the number of stored proxels increases to two third of the capacity. Increasing array's capacity includes creating a new array whose capacity is twice as that old array's capacity. Then we copy all the proxels to the new array and we drop the old array.

While copying the proxels to the new array, the key is recalculated according to the capacity of the new array. The proxels are placed according to the newly computed index. Therefore the time required for copying each proxel will be O(n) where n is the number of proxels in the old array. Proxels of very high number such as 10E+3 to 10E+6 occurs frequently in this approach. We also have additional key computation time for each proxel. Even though it supports searching a proxel in single access, the time required for copying and recalculating the key affects the running time of this approach to a very large extent. In particular when the number of proxels is very large.

Multidimensional array

The fastest search for a proxel can be obtained in this approach. We don't need to compute a key for a proxel. Consider the study model Petri net, we have to declare a 5 dimensional proxel array. One for the marking and other four for the transitions. If there is a proxel containing marking m_0 and age intensity vector (2dt,4dt,0,0), then we can search it, by accessing the location, proxelarray[0][2][4][0][0]. If it is a proxel with marking m_2 and age intensity vector (0,0,5dt,0), then we can search it by accessing proxelarray[2][0][0][5][0].

The disadvantage of this approach is, the wastage of memory space. Except for the first entry where the index is the marking number, we have to declare a large dimension for other four entries. This declaration should be large enough, to hold all the intensity values. Further, we don't use all the

declared locations. It is apparent in our study model, that for the marking m_0, [0][*][*][0][0], the last two index will be always zero. This makes the matrix very sparse resulting in higher wastage of memory.

Binary tree

The binary tree with balancing supports the proxel search in O(log N), even in the worst case. This is discussed in the earlier sections. Balancing takes longer time and reduce the performance of this simulation approach to a large extent. Hence the current implementation uses the unbalanced binary tree. It performs well for the best case with O(log N) but the worst case for storing or searching takes O(N).

Further, the important thing to notice is system over head. We use recursion algorithms to process the proxels in a binary tree [Traversing binary trees]. This recursion consumes systems stack memory, especially when the number of proxels is very large. In the worst case scenario, the system memory is occupied with the address of *(n-1)* proxels. When this n is large, this approach consumes lot of *stack memory* [David Eck and Bradley Kjel 2004]. This affects the processing time of the proxels and the simulation approach significantly.

Hash table

Hash table uses the hashing approach described in the fundamentals section. Searching a proxel in the hash table takes single access if there is no collisions. But the worst case runtime is O(N). This occurs when the hashing algorithm maps all the keys to the same index.

Efficient usage of hash table lies in the design of the key indexing algorithm. The algorithm aims to distribute the keys evenly. The distribution here refers to lesser number of collisions. The worst case behavior for this strategy is same as that of the unbalanced binary tree. Further it is practically very difficult to design a hash algorithm without collisions.

3.2.2 Storage approach based on array and hashing

This storage strategy aims to reduce the run time and memory requirement for the proxel based simulation. The first part describes the design and the second part deals with the operations. We use the same Petri net shown in the Figure 3.1 for illustrating proposed implementation.

Storage Design

The implementation of the proposed approach is based on the array and hashing techniques. First we define an array with capacity equal to the number of markings. The index i of the array corresponds to the i of the marking m_i. We name this array as *marking array*. Each entry for this array contains another array of capacity two. We call this array as *time step array*. The index 0 in this array contains the proxels for the current time step. The index 1 of the *time step array*, is used to store the proxels generated for the next time step. We define index 0 as $i_{current}$ and 1 as i_{next} similar to $BT_{current}$ and BT_{next} in the binary tree of the current implementation. The index $i_{current}$ and i_{next} hold another array respectively. This is used to store the proxels. We name this array as *proxel array*. The initial proxel is stored in the proxel array of $i_{current}$ whose marking array's index is zero. This is because the initial proxel contains the marking m_0.

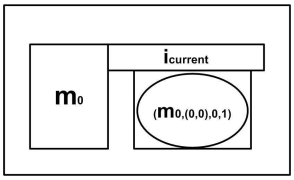

Figure 3.4: Storage structure at time step 0

Estimating proxels for next step

Before starting the proxel generation, we calculate the number of proxels generated for the next time step from the reachability graph. In our case, we have one proxel containing marking m_0. Therefore, next step will have two proxels containing the marking m_1 and m_2. Also we have another proxel which contains the same marking m_0. We declare the proxel array in i_{next} for each marking with this estimation. In other words it possible to predict the number of proxels generated for each marking for the next time step. This number comes from the number of transitions enabled in the proxel's marking.

We use this estimation as the capacity of the proxel array for each marking in i_{next}. We start processing the proxel in $i_{current}$ for each marking. We have only the initial proxel we process that proxel. This is shown in the Figure 3.5

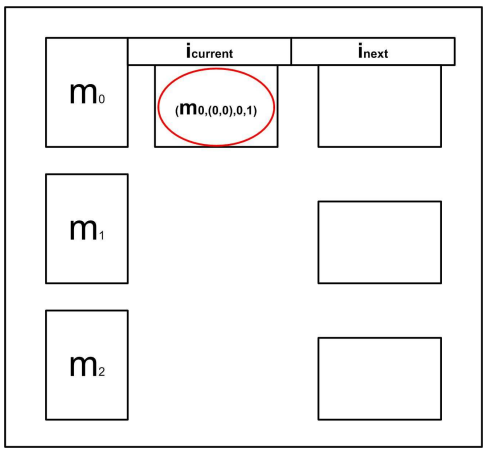

Figure 3.5: Proxel estimation

Key Computation

The proxel which contains the marking m_1 is processed. Updating the age intensity vector is done according to the transitions enabling conditions and memory policy. The key for the proxel is computed only with the age intensity vector. The data structure place the proxel in the corresponding marking array which makes the key computation free from marking. The implementation of the proposed approach uses the CBU hash function. This computes the hash value x_i, from the proxel key (τ_i). The index of the proxel array is computed from the hash value by the following function:

$$h(\tau_i) \quad = \quad (x_i) \quad mod \quad N_i \tag{3.1}$$

The proxel is stored in the computed index. The N_i used in the hash function is the estimated capacity of the proxel array, for the corresponding marking m_i, in i_{next}. The above process is repeated for the proxel containing marking m_2 and the same marking proxel m_0. This is shown in the Figure 3.6.

Proxel search

Given a proxel, searching takes single access. The index of the proxel array is computed by the hash function with the age intensity vector. The index of the proxel array in i_{next}, of the corresponding marking array is searched. If a proxel is there with same age intensity vector, then the probabilities are added to the existing proxel. Otherwise the proxel is stored in the computed index of the proxel array. Therefore in best cases, runtime for searching a proxel in the proposed implementation takes O(1).

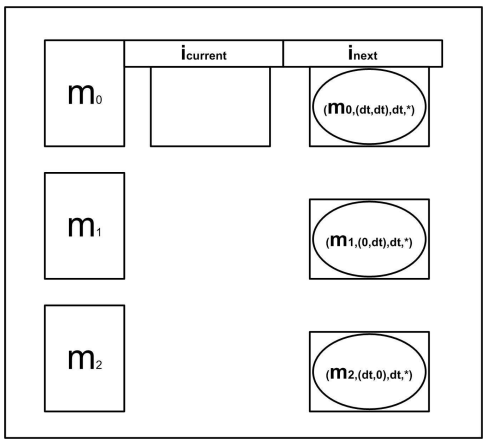

Figure 3.6: Storage structure at time step dt

Simulation

After the above steps, we have the probabilities of different markings after first time step. The steady state probability or the probability at the end of the specified time, for each markings of the Petri net, is obtained from this simulation approach as follows:

For each step, process all the proxels in the proxel array of $i_{current}$. Once this process is over then remove the proxel array in $i_{current}$ and make the i_{next}'s proxel array as $i_{current}$'s proxel array for the next time step.

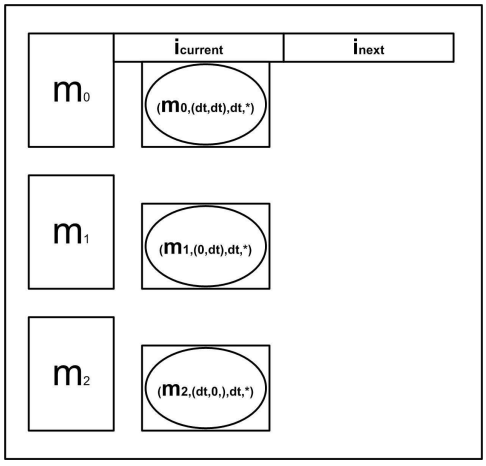

Figure 3.7: Interchanging the proxel arrays

For next step, repeat the procedures followed from *"proxel estimation"* to *"simulation"* until specified time has been reached.

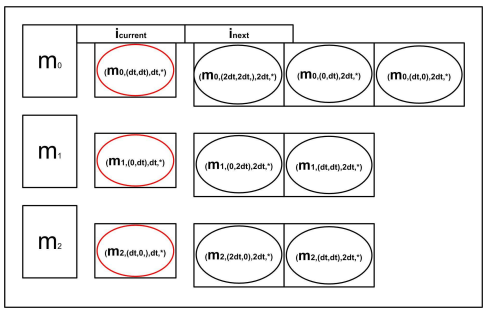

Figure 3.8: Storage structure at time step 2dt

Collision handling

The proxel search method has a best case running time of O(1). This is possible when there are no collisions. This is shown in the Figure 3.8. Collisions refers to two or more proxels containing same marking but different age intensity vector with same hash value. Therefore we are forced to store them in the same index of the proxel array. A bin is created in the collision index. This bin is also an array. The proxels having same computed index for the proxel array are stored in the bin. This is shown in the Figure 3.9

The proposed implementation adopts linear probing technique for searching the proxels in the bin. This has the following steps

1. probing or pruning each collision proxels by checking its age intensity vector.

2. If one such proxel exist during the step (1), whose age intensity vector is same as that of the searching proxel, then the probabilities are added.

3. If step (2) is not satisfied then the searching proxel is placed at the end of the bin.

In worst cases, time taken to search a proxel is O(n), where n is the number of collision proxels. This n is a number of proxels which belongs to the marking of the marking array. But in the unbalanced binary tree and hash tables, O(n) corresponds to the total proxels stored in the data structure. But still if we don't have any collisions, the best case running time for the proposed approach is O(1).

While processing the proxels in the proxel array of $i_{current}$, the proxels stored in the bin is processed one by one.

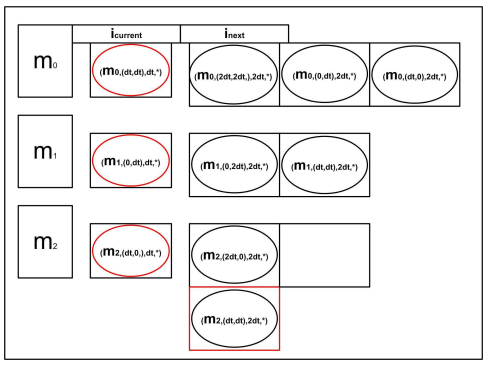

Figure 3.9: Collisions in the storage structure

3.2.3 Role of hashing in the proxel search

The worst case runtime for searching a proxel is proportional to the number of collisions which is based on the efficiency of the hashing method employed for hashing the key. The runtime of the hashing method is proportional to the length of the proxel key. The implementation of the current approach uses the marking and the age intensity vector as the key. For an example, if we have a proxel specified as,

$$\mathrm{P}=(\ \mathrm{m}_0,(\tau_1,\tau_2,\tau_3,\tau_4),t,p_r\ )$$

The key is computed by combining m_0 with τ_1,τ_2,τ_3,τ_4. But in the proposed implementation, we use only (τ_1,τ_2) as the marking dependent age intensity vector for m_0. The proposed design has separate proxel array for each marking. Therefore the implementation needs (τ_1,τ_2) as the argument for computing the key. Hence we reduced three entities for the key computation. This reduces the run time of the whole approach significantly. In particular, when the number of transitions is large and our nearly optimal supplementary algorithm removes more irrelevant entries.

It is nearly impossible to design a hashing technique, with one to one relation. This relation maps different proxel key to different value. In other words, it is very hard to design a hashing technique, without collisions. The design of the algorithm lies in reducing the collisions. Reducing the collisions, increases the time complexity of the hashing technique. If the collisions are very less then proxel search can be achieved in *nearly* O(1). This reduces the proxel search time and the run time of the simulation approach. But increase in the time complexity makes the hashing technique expensive in terms of running time. We reduce the proxel search time but this makes key computation time expensive. Hence a trade off is necessary between the time complexity of the hashing technique and proxel search

time. We adopted the CBU hashing algorithm based on these trade off in the proposed implementation.

3.2.4 Significance of the proposed storage design

With the hashing based array implementation, we reduce the key length by removing the marking from the key. We include only the reduced age intensity vector. The number of bits in the key is reduced. This makes the key computation faster. This faster key computation makes the proxel key comparison faster. Further we use hashing technique to store the proxels. In best case, we can search a proxel in a single access. Even in the worst case, we search proxel in a linear time O(n). This n includes only the number of proxels containing same marking. The binary tree implementation in the existing approach has a best case running time of O(log N) and worst case of O(N). Here N refers to the total number of proxels, irrespective of marking in the data structure.

Approach	Best case	Worst case
Existing approach	O(log N)	O(N)
Proposed approach	O(1)	O(n)

3.3 Evaluating the proposed approach

We suggest two ways for improving the current implementation. The preprocessing of the Petri nets makes the reachability graph generation an automated process. Further it gives a marking dependent age intensity vector. This approach reduces the memory and run time requirements for the proxel simulation approach. Therefore we evaluate this preprocessing approach with runtime and memory in the experimentations.

The second way is an improvement concerning the usage of data structure in the current implementation. The word *"improvement"* here again refers to runtime and memory requirements. This proposal aims to reduce the proxel search time and the amount of memory utilized. Therefore we evaluate the storage design with the runtime and memory. The important factor which affects the proposed implementation is the number of collisions. We also evaluate the proposed approach with the number of collisions.

3.4 User Interface

We designed a user friendly interface for the implementation of proxel based simulation method. We also present the simulation results in a graphical format. This enables the user to visualize the behaviour of model over time.

Here we describe the design procedures, modules and instructions for using this interface.

3.4.1 Design of the interface

The interface has three modules. Java Swings [Java Swings examples 2004] is used for the implementation. We present these modules and afterwards we describe their construction. The modules are named according to the function.

1. Preprocessing module
2. Simulation module
3. Results visualization module

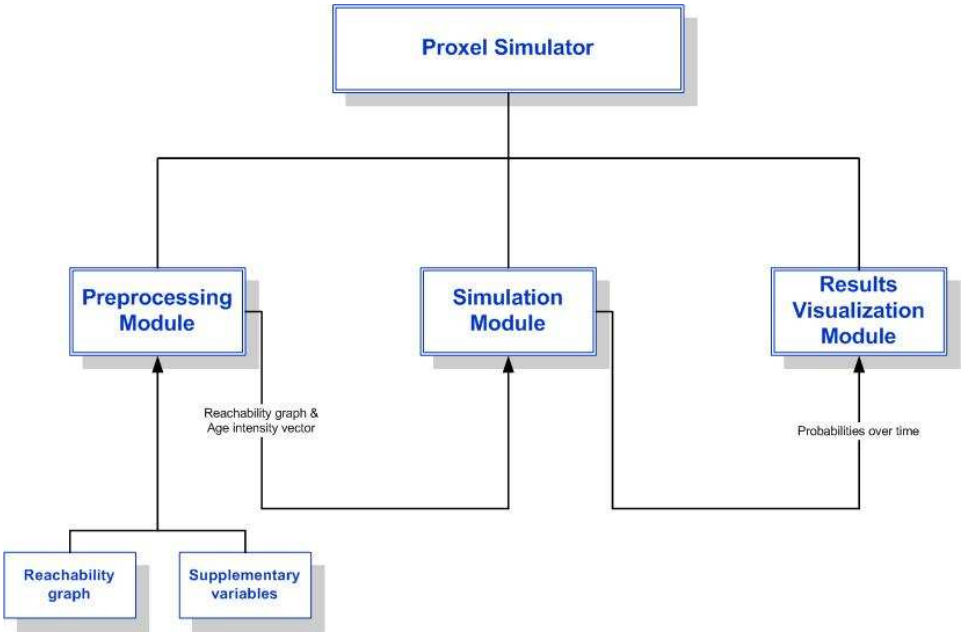

Figure 3.10: Software modules for proxel based simulator

Preprocessing module : The function of this module is to preprocess the Petri nets. As we have described in the previous sections, this process has two functions. First we generate reachability graph from the Petri net afterwards we find the marking dependent age intensity vector. Therefore we designed two functions for these process.

The function,*"reachability graph generation"*, will take the input Petri net. The model is exactly defined by the specification file for a Petri

net. We extract those information from input specification and store them in the appropriate data structures. The data structure is used in a way such that the preprocessing does not take longer time for this process. This function includes the reachability algorithm for generating reachability graph from the Petri net in the form of reachability matrix. Another function finds marking dependent age intensity vector. It will take the reachability matrix generated in the above step. This function includes the algorithm to compute nearly optimal supplementary variables. Finally it define the marking dependent age intensity vector.

The final output of this module is composed of results from the two functions along with some specifications from the Petri net. These specification includes discretisation parameter *dt*, simulation time t and the details about the transitions.

Simulation module We take the output from the preprocessing module here to start the simulation. We keep track of the probability values of each marking at every time step. We store these values in a separate text file. We do this for two purposes. First we can use this text file for later analysis. Next, we are going to visualize the results, in particular the probability values for each marking at each time step based on this file.

Results visualization module The main aim of this module is to visualize the behavior of the model over time. We present this in a two dimensional graph, where the x axis corresponds to the simulation time and y axis corresponds to the probability values for each marking. We also provide this visualization in a JPG picture format [Java Graph 2004] . For later analysis, we not only have the text file with probability values but also an image showing its behavior.

3.4.2 Instructions for usage

In this section we provide the instructions to use our interface. The Figure 3.11 shows a screen shot of our interface. We have six buttons at the top. The first button "*Open Petri net*" is a file browser which is used to specify the location of the Petri net file. The second button "*Save Reachability*" is used to specify the location where the reachability graph of the petri net should be stored. The third button "*Generate*" will generate the reachability information and store in the file specified in the second menu. This information also contains the marking dependent age intensity vector. The fourth button "*Save Results*" is used to create a file which stores the

probability values for each marking. The fifth button "*Simulate*" starts the simulation process. When the process is completed, we inform the user with a pop up message. The sixth menu "*Graph*" will visualize the probability results form the text file where we store the probability values. We also provide a button at the bottom of the interface called "*Auto Simulation*". This is a button which automates all of the six button's process. We specify only the location of the Petri net file. It will shows the user with the probability visualization, for each marking. We use temporary files to hold the intermediate values.

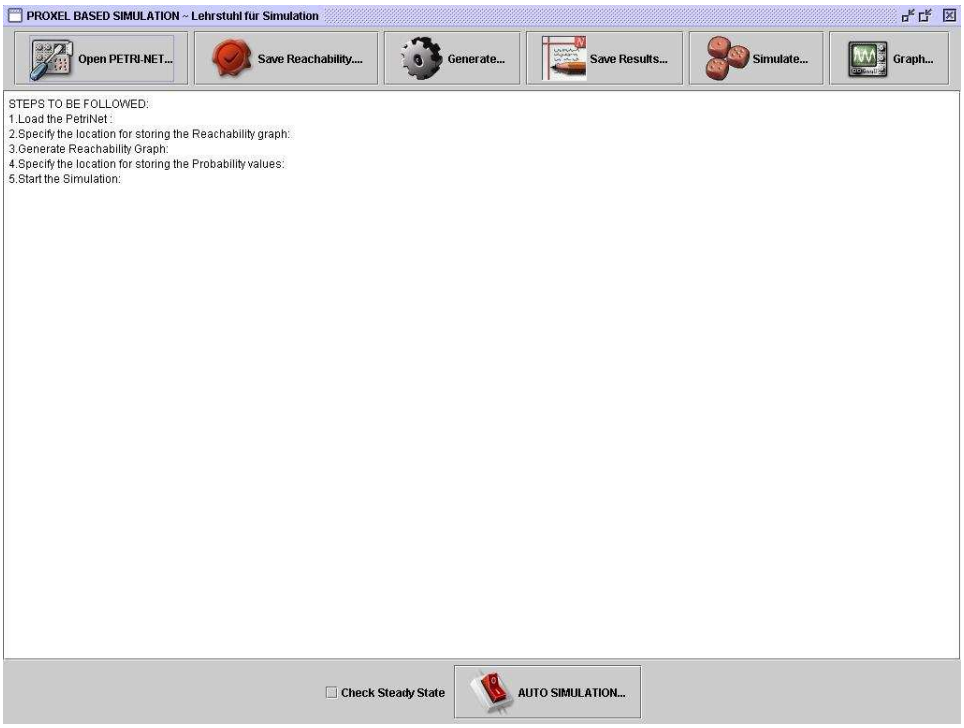

Figure 3.11: Proxel based Simulator

Figure 3.12 illustrates the visualization of probability values. There is also the marking interpretation at the bottom of the graph.

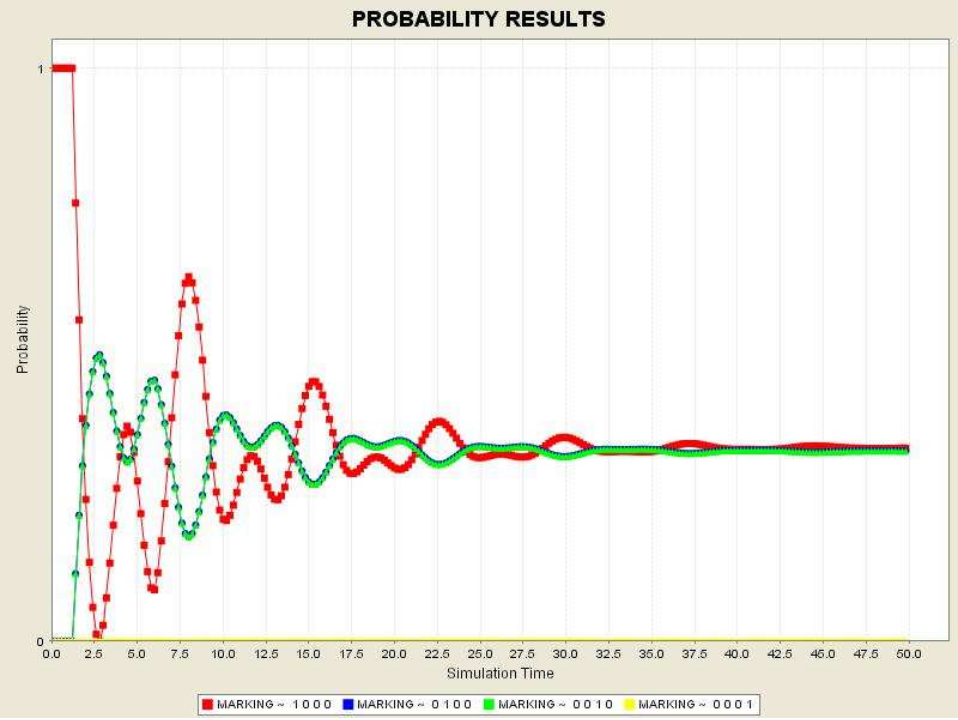

Figure 3.12: Proxel based Simulator

3.5 Overview of the proposed approach

We are using stochastic Petri nets to model the real word entities. We define the specification of Petri nets. Afterwards we construct the reachability graph for the Petri net, which consists of possible discrete states of the model, in the form of markings. After extracting this information we use an algorithm to compute the age intensity vector for each marking. This algorithm is based on the guidelines which we have defined for this purpose. Therefore we refer this process as, finding nearly optimal number of supplementary variables for each marking. With these information we start our simulation approach. We designed an improved storage strategy based on the hashing and arrays for proxel based simulation. The implementation aims to reduce the time required for searching a proxel in the data structure. Finally we present the simulation results in a user friendly way. The proposed implementation has its own limitations. They are stated as follows:

Petri net specification

The implementation of the proposed approach works on the Petri net specification file. We defined the model definition. This is not a standard defi-

nition. We need a common information interchange format. This format is independent of specific tools and platforms. If we can read this format, in our approach, then we do not restrict the user to know our definition. Petri Net Markup Language can be used for this purpose.

Boundedness

The reachability graph is constructed based on the assumption of the bounded condition of the Petri net. Petri net is said to be bounded, when the set containing all the possible discrete states, reached by the model is finite. In other words, the markings generated by the reachability graph is finite.

This forms the second limitation. We are starting the simulation after computing the reachability graph. If the markings are infinite, then we cannot start our implementation.

Optimality for supplementary variables

We defined our own guidelines for the optimality of supplementary variables. Our algorithm to find marking dependent age intensity vector works only for these guidelines. Hence we call this process as finding nearly optimal number of supplementary variables for each marking.

These guidelines give rise to third limitation. We are able to compute age intensity vector based only on these guidelines. Still we have supplementary variables which are not relevant to the corresponding markings in the age intensity vector.

Level of vanishing markings

During simulation, if we encounter a proxel containing a vanishing marking then we find the successor proxel containing tangible marking. We have this as a single level. We assume that a vanishing marking's successor is always a tangible for the proposed implementation.

We have our third limitation as lack of support for multi level vanishing marking. This multi level refers to the situation in which a vanishing marking has a series of vanishing marking to reach a tangible marking.

Proxel estimation and computation time

Consider the following scenario during proxel processing. Suppose the capacity is estimated as N, in the previous step. The data structure stored only (N-k) proxels in the proxel array. That is k proxels were combined,

during the proxel generation process. We iterate N times during the next step to process proxels in the proxel array. While iterating, if we encounter a proxel then we will process, otherwise we move to the next iteration. The same case applies to proxel threshold and collisions.

This forms our fourth limitation of the proposed approach. In particular if the proxel combination, threshold and collisions are more, we waste computation time for simply iterating over the empty index of the proxel array.

Proxel estimation and memory requirement

During simulation, we estimate the number of proxels for the next step. For example, if there are three markings and we estimated N_0, N_1, N_2 proxels for each marking. This N_i will be the capacity of the proxel arrays. We allocate memory for three proxel arrays with N_0, N_1, N_2 as the capacity of the arrays. All the allocated memory is not used by the proxels. Some of the memory space is left unused. This comes from three conditions. Proxel combination, threshold and collisions. The collisions not only leave a memory space but also creates new space in the bin.

This forms our fifth limitation. We are unable to estimate the exact number of proxels. The proxel collision and threshold conditions are difficult to predict. On the other hand the proxel combination is possible to know in advance. But our approach does not have this functionality.

Chapter 4

Experimentations and Results

In this chapter we present experimentation results of our implementation. The experiments related to preprocessing Petri nets compare the consequences of preprocessing in proxel based simulation. The experiments related to storage strategies compare the effects of the different data structures used in proxel based simulation. The experiments are carried out with the following setup.

- CPU-Mobile AMD Athlon^{TM} XP 2000+ of speed 1.67 GHZ
- 512 MB RAM
- Microsoft® Windows® XP operating system
- Java 1.4.2
- JVM memory 320 MB
- Stack size of 40 MB for Binary tree implementation.

The experiments are classified as follows:

- Preprocessing experiments which analyze the performance criteria, for the array with hashing and the binary tree implementations
- Storage strategy experiments which analyze the performance criteria, for the different implementations
- Experiments which measure the performance criteria, of the current approach(Binary tree without preprocessing) with proposed approach(array using hashing with preprocessing)

4.1 Preprocessing

The effects of preprocessing in the proxel based simulation can be measured with computational time and the memory utilization. The experiments in this section are based on analysis of these parameters with respect to storage strategies, which includes the implementations of the binary tree and the proposed data structures. The Petri net used for the experimentations is shown in the Figure 4.1. The transitions are described below.

- T_1 and T_2 are uniformly distributed in the interval (1.5,3.3) with age policy
- T_3 and T_4 are also uniformly distributed in the same interval (1.5,3.3) but have enable memory policy.
- T_5 is deterministic with 0.3 and contains enable memory policy
- t=100 and dt=0.5

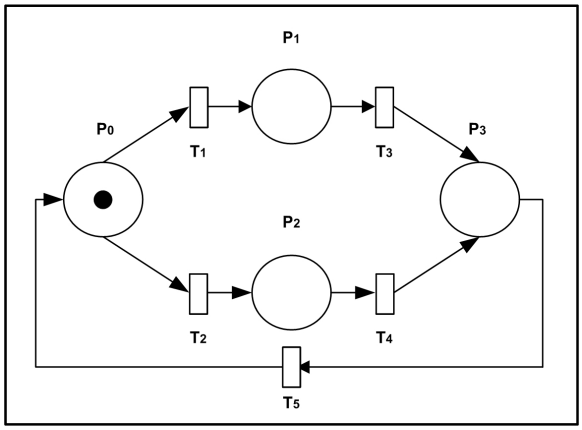

Figure 4.1: Petri net used for experimentations

4.1.1 Binary tree

The results from the Figure 4.2 compares the computational time of the binary tree implementation for proxel based simulation approach, before and after preprocessing. We plot the values of simulation time against the runtime.

The curve "*constant size*" shows the runtime without preprocessing. The age intensity vector includes an entry for each transition in the Petri net. The Petri net used in this experiment has five transitions. Therefore the

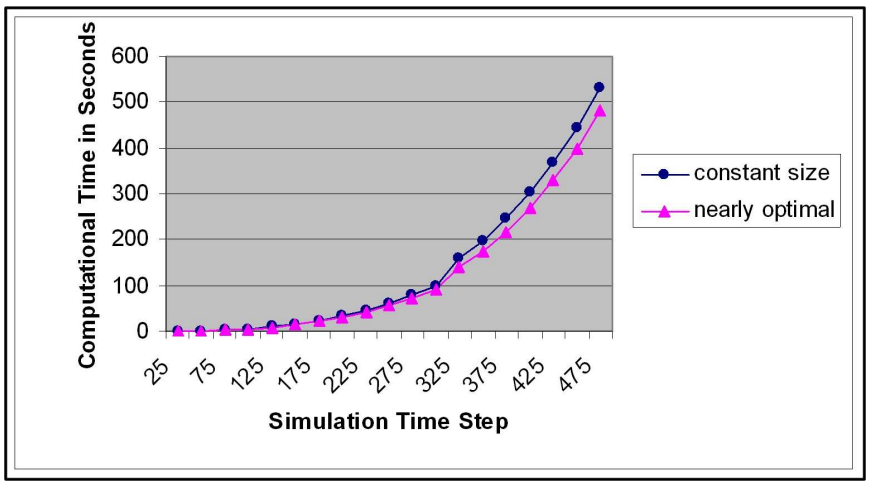

Figure 4.2: Effect of preprocessing in runtime of a Binary tree

number of supplementary variables in the age intensity vector is always five. After preprocessing the Petri net, algorithm which computes the nearly optimal supplementary variables removes three irrelevant variables from each marking. The runtime, after preprocessing is shown with the curve "*nearly optimal*". From this experiment we conclude that, our preprocessing proposal improves the runtime of the current approach although not significantly.

The reasons for the above mentioned results are stated as follows. The binary tree implementation stores the proxels by computing the key. The key of a proxel is computed from the marking and the age intensity vector. The key computation time is proportional to the length of the key. The length of the key decreases with decrease in the number of entries in the age intensity vector. This reduces the key computation time. The search in the binary tree, for the occurrence of a newly generated proxel needs the comparison of the keys which means comparing each bit of the new proxel's key with the proxels in the binary tree. The time needed for this is proportional to the length of the key. After preprocessing, the key length is reduced. Thus it improves the computational time of the approach.

The results from the Figure 4.3 compares the memory usage of the proxel based simulation approach before and after preprocessing. We plot the simulation time step against memory allocated for each step.

This results shows that, after preprocessing there is a significant improvement in the memory requirement. The curve "*nearly optimal*" has lower memory requirements then the "*constant size*" one.

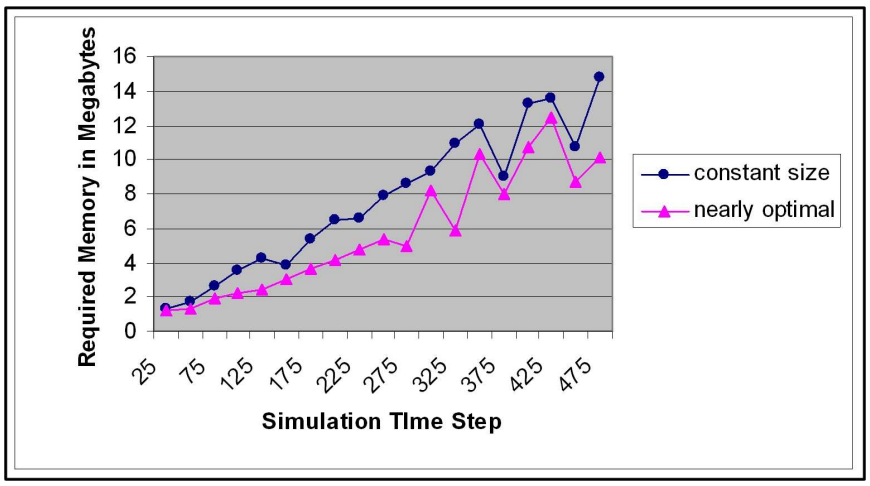

Figure 4.3: Effect of preprocessing in memory requirement of a Binary tree

The reason for this improvement are stated as follows. By removing each irrelevant supplementary variable in the age intensity vector of the corresponding marking, we save 32 bits for each proxel. The age intensity is a float value. Java allocates 32 bits for each such entry. We used the same Petri net involved in the run time experiment. The preprocessing removes three entries from each marking. So we saved 96 bits for each proxel. When time increases, the number of proxels generated are more. Therefore there is significant reduction in the memory.

4.1.2 Array with hashing method

We carried the same experiments explained in the previous section, for the proposed implementation of the array based on the hashing technique.

Figure 4.4 shows the computational time before and after preprocessing. We use the same terminology, "*constant size*" and "*nearly optimal*" for the two approaches. We use the same Petri net used in the previous section for this experiment.

From the results, after preprocessing, the computational time for the simulation approach is reduced. The reason for this improvement is same as explained in the previous section.

Figure 4.5 shows the amount of memory required for the array based implementation of the proxel based simulation, before and after to preprocessing.

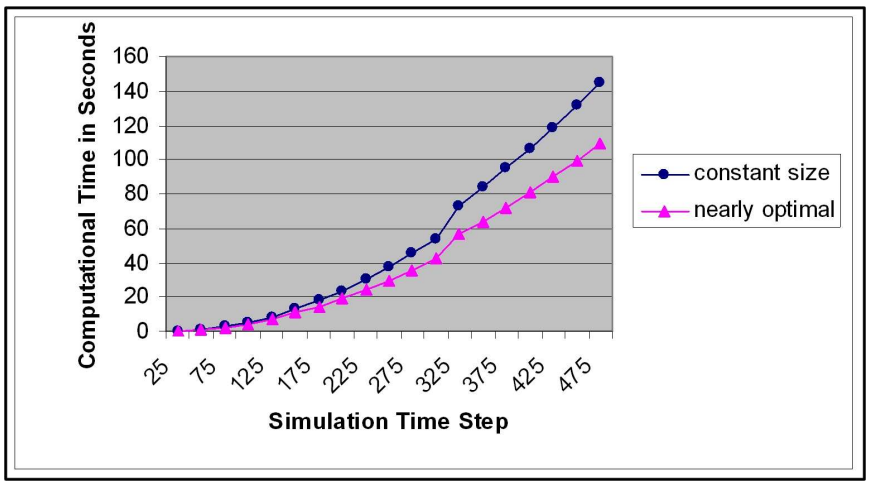

Figure 4.4: Effect of preprocessing in runtime of the array

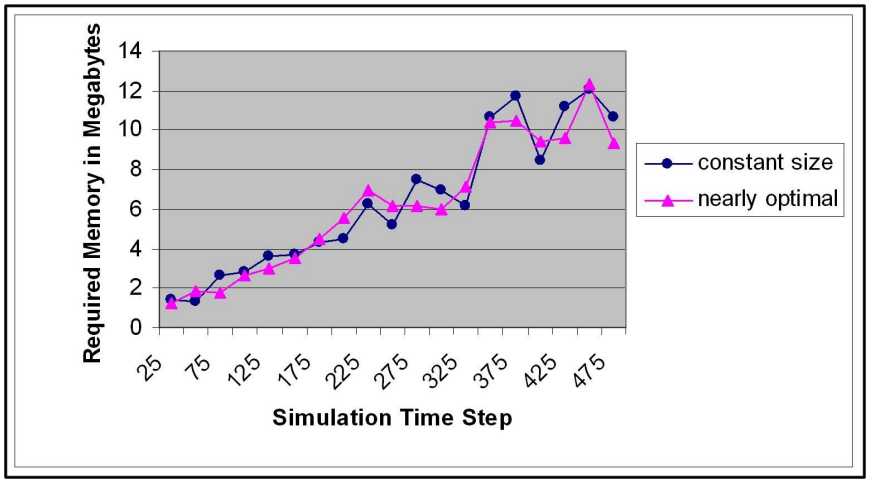

Figure 4.5: Effect of preprocessing in memory requirement of the array

The result which we got for this experiment, is not one which we expected. We expected the memory requirements after preprocessing, will be always less than its competitor. In order to understand this unexpected behaviour, the number of collisions is analyzed for each time step. This is shown in the Figure 4.6. We plot the number of collisions for each time step during proxel based simulation, after preprocessing.

The key of a proxel is computed from the age intensity vector. We use the hashing technique to compute the index from the key. The proxel is stored in the computed index. Collisions are caused by two or more proxels

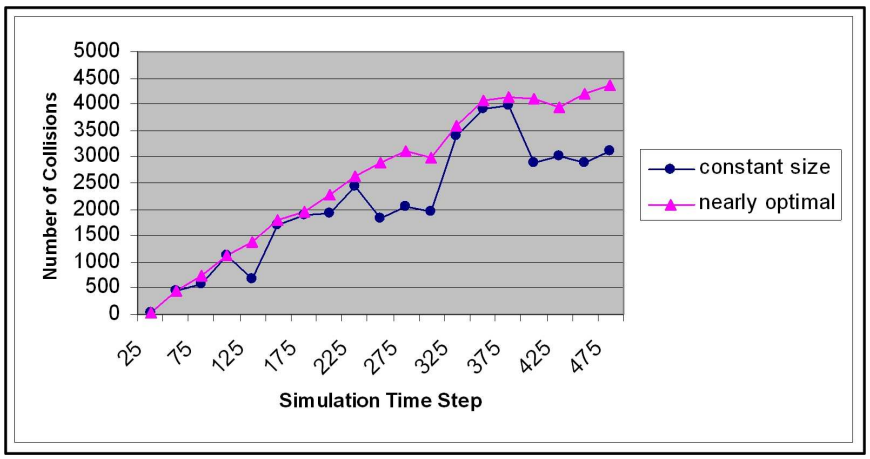

Figure 4.6: Effect of preprocessing in collisions of the array

with same computed index but different age intensity vector. This collision depends on the key length and the estimated capacity (N_i) of the proxel array. Reduction in the key length increases the number of collisions. For each such collision a new space is created in the bin of the proxel array. The collision proxels are stored there. At the same time the position of the proxel in the proxel array is vacant. Larger the collision, larger the unused allocated memory will be . This makes the memory requirements larger. This is clear if we analyze time steps form 225 to 275. Here the number of collisions are fewer without preprocessing. The key consists of five supplementary variables from the age intensity vector. Hashing technique produces less number of collisions for the key containing five supplementary variables compared to the key, which contains two supplementary variables. If we analyze the same time step intervals form 225 to 275 in the Figure 4.5, the memory requirement is smaller for "*constant size*" compared to "*nearly optimal*". The same argument applies for time steps from 425 to 475. At these time steps, the unused allocated memory is more when compared to the saved memory. The memory saving comes from the removal of irrelevant supplementary variables.

The irrelevant supplementary variables in the age intensity vector increases the run time but reduces the number of collisions during proxel storage. The Figure 4.7 shows the results of runtime for the proxel based simulation with respect to the number of irrelevant supplementary variables such as three, four, five, six, seven for each marking.

On an average if there are three irrelevant supplementary variables in each marking, its computation is more than that of the nearly optimal case. Figure 4.8 shows the memory requirements for each of the case. The irreg-

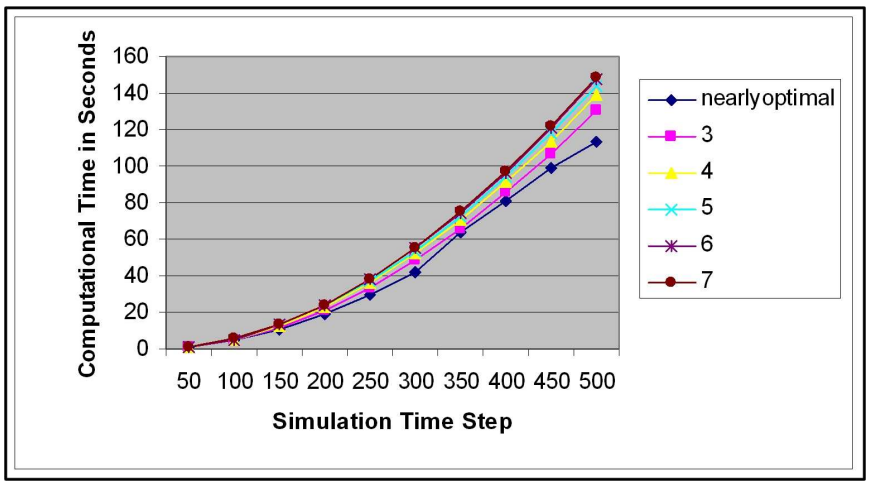

Figure 4.7: Effect of irrelevant supplementary variables in runtime.

ularity in the memory requirement comes from the number of collision of the hashing technique. Higher the collisions, larger the amount of unused memory is.

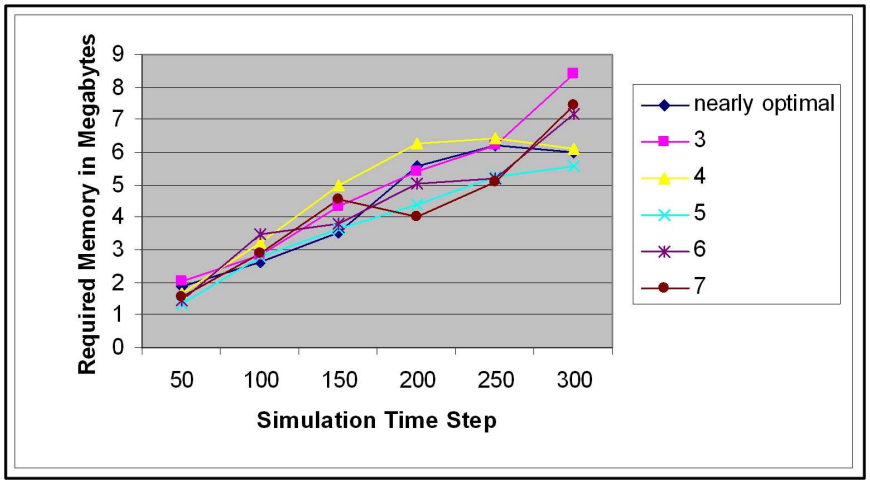

Figure 4.8: Effect of irrelevant supplementary variables in memory.

We are using *object data type* in Java (equivalent to *pointers in C*) for storing the proxels. The amount of memory required for these data types are not the amount of memory utilized. Therefore the required memory is not the used memory. The required memory is less after preprocessing and increases with the number of irrelevant supplementary variables for each marking. The amount of unused memory will be reused again [Java HotSpot VM Options 2004].

Our proposed storage strategy behaves in the same way as that of the binary tree. The key computation time is made shorter with preprocessing. The memory requirements varies with the number of collisions. The length of the key determines the number of collisions in this case. The smaller number of bits in the key, then greater the number of collisions and the memory requirements are. The same scenario decreases the key computation time and the whole computation time of the simulation approach.

4.2 Proposed storage strategy

This section describes the experimentations related with the proposed storage strategy. The runtime and the memory requirements of the array implementation with hashing method are measured and compared with the binary tree implementation, for evaluating the proposed design.

4.2.1 Key computation

The index for storing a proxel in the proxel array is computed form the key. The array implementation use only the age intensity vector. The supplementary variables constitute the age intensity vector. We use hashing technique for indexing the key. The primary goal of a hashing technique is to reduce the number of collisions. The complexity of the hashing technique is measured in terms of the number of machine instructions required internally for the technique. In order to find the best hashing technique for the array implementation, We analyzed the following hash techniques [Hash Functions for Hashtablelookup 2004].

- CBU hash
- Java hash
- Additive hash
- Rotative hash
- One-at-a-time hash
- 64 bit-BUZ hash [64-bit Hash function].

Figure 4.9 shows the results of the number of collisions for each time step during proxel based simulation, with respect to different hashing techniques. From these experiments we conclude that increasing the complexity of the hashing technique, decreases the collisions of the proxels. CBU hash

is the simplest of hashing method mentioned above. Therefore the number of collisions is higher for this method.

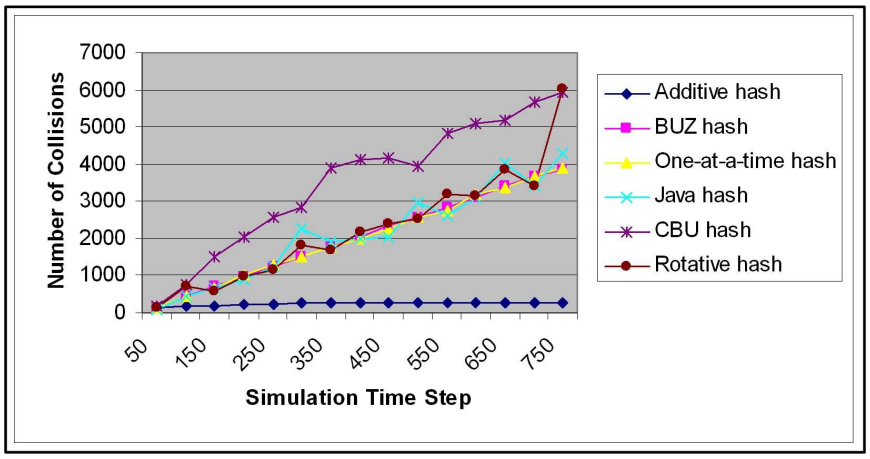

Figure 4.9: Proxel collisions for different hashing techniques.

Figure 4.10 shows the runtime of the simulation approach with respect to different hashing techniques.

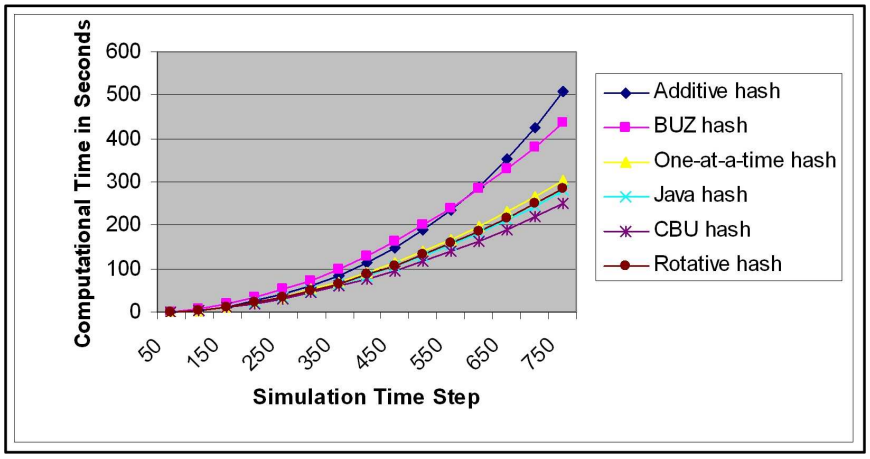

Figure 4.10: Simulation runtime for different hashing techniques.

From these results we conclude that increasing the complexity of the hashing technique, increases the simulation runtime. The higher runtime is due to the higher key computation time. A complex hashing method takes more time for index computation from the key. This produces well distributed index from the keys which results in minimum number of collisions at the expense of larger computation time. Further from these experiments

we observe that, time taken for linear probing the collision proxels is less when compared to the time taken for producing the well distributed index form keys.

4.2.2 Comparing storage strategies

Before preprocessing

Figure 4.11 shows the comparison of runtime with respect to the binary tree and array implementations, without preprocessing. The size of the age intensity vector is the same for both strategies and it includes all of the transitions of the Petri net.

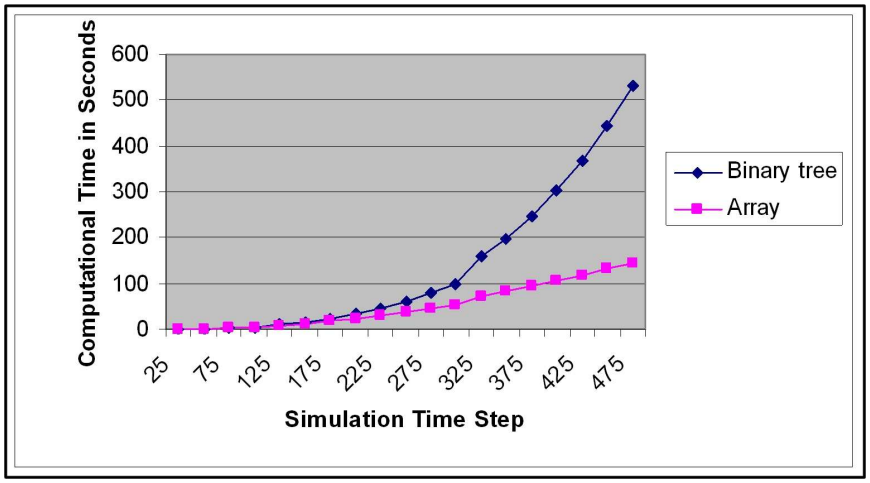

Figure 4.11: Comparison of storage strategies with respect to runtime.

From these results we observe that, implementation of the proposed storage strategy based on array and hashing for proxel based simulation has shorter computation time, when compared with binary tree implementation. In the best case, the proxel search takes single access in the array based implementation. The binary tree for the same case, the searching takes logarithmic time.

Consider the worst case scenario. For the array based implementation all the proxels are indexed to the same location of the proxel array. The bin contains all the proxels of the marking m_i. Therefore the proxel search will take $O(n_i)$ where n_i refers to the number of proxels stored in the proxel array of the marking array m_i. The binary tree implementation for the same case, all the proxels are always added either as left child or right child. If there is 1000 proxels with the searching proxel as the lower most proxel,

then the search will take 1000 access to reach that proxel. Therefore the proxel search takes O(N) where N refers to the total number of proxels in the tree.

The results for the amount of memory required for different implementations are shown in the Figure: 4.12. It can be seen that the array implementation has lower memory requirement than the binary tree implementation, most of the time.

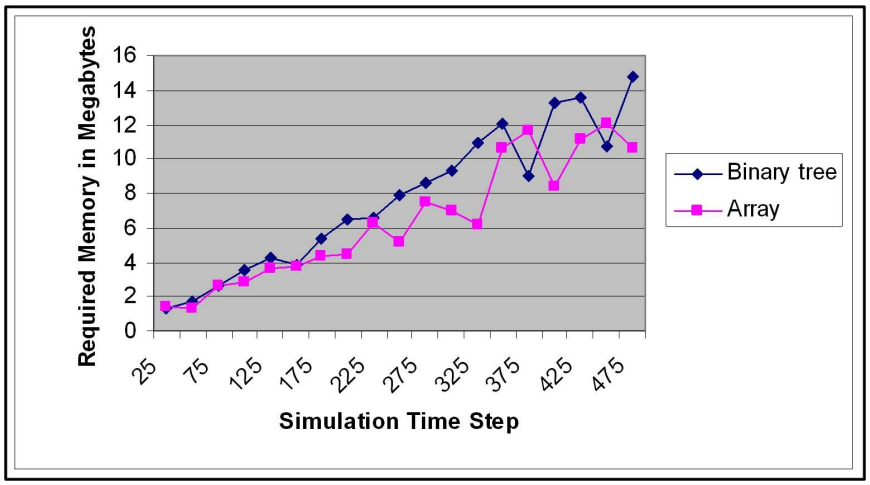

Figure 4.12: Comparison of storage strategies with respect to memory requirement

In the array implementation, after processing a proxel, it is removed from the array. The binary tree keeps the proxels until all of them are processed. Removing a proxel causes rearrangement of proxels in the tree which increases the simulation time.The proxel generation increases the memory requirements in both array and binary tree implementations. The memory is freed after processing a proxel in the former but not in the later.

After preprocessing

Figure 4.13 shows the computational time of the proxel simulation approach, after preprocessing with respect to two storage strategies. Each marking of the Petri net is associated with marking dependent age intensity vector. The reasons for shorter computation time of the array based implementation is same as explained in the previous section. The proxel search takes less time compared to binary tree.

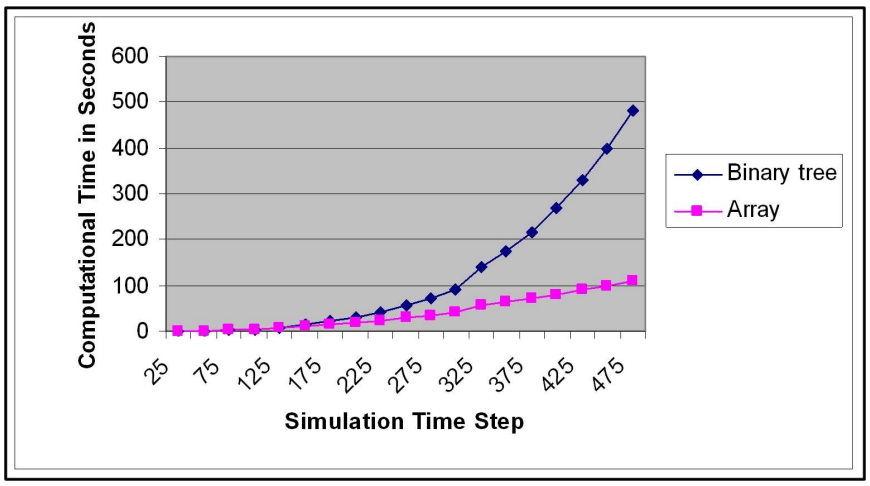

Figure 4.13: Comparison of storage strategies with respect to runtime.

Figure 4.14 shows the memory requirements for same case. The array based implementation has higher memory requirements because of the following reasons:

Proxel collisions: Collision proxels create new space in the colliding bin and leave space in the memory. They do not use the memory allocated in the proxel array.

Proxel combination: This creates the over estimation of the number of proxels. If a proxel combines with another proxel by adding its probability, then the corresponding space in the allocated memory is void.

Probability threshold: The approach discard proxels with probabilities less than the threshold value. This also creates unused memory in the same way as mentioned above.

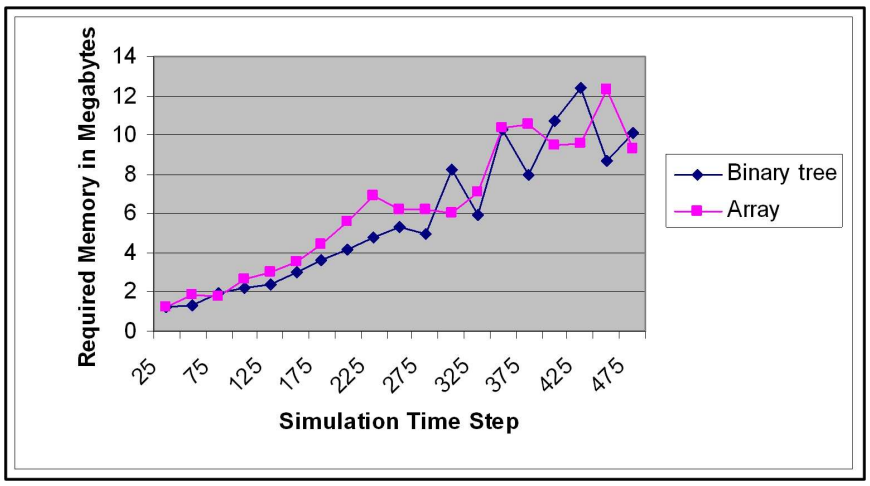

Figure 4.14: Comparison of storage strategies with respect to memory requirement.

4.3 Existing approach vs Proposed approach

This section compares the pure old approach with the proposed new approach. The former uses the binary tree implementations for proxel storage without preprocessing. The later uses the array implementation for proxel storage with preprocessing.

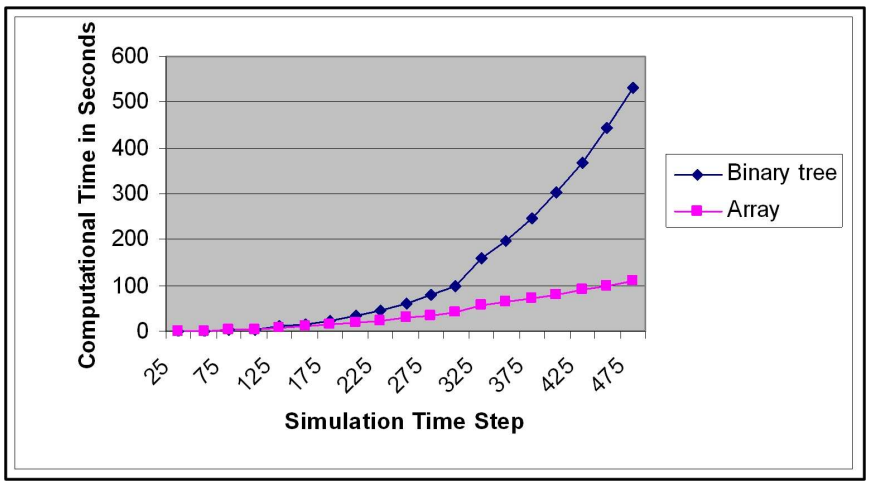

Figure 4.15: Comparison of existing strategy with proposed strategy based on runtime

Figure 4.15 shows the computational time taken for the proxel based simulation with respect to the two implementations. The results revel that

the runtime of the array implementation is shorter than binary tree implementation. This is for two reasons. First one is shorter proxel search time, which is discussed in the previous section. Second one is reduced key computation time. The key for the proxel consists of only the nearly optimal supplementary variables. In the current implementation, the key contains of all the supplementary variables and the marking. Increase in the key length increases the key computation time and key comparing time during proxel generation and search respectively.

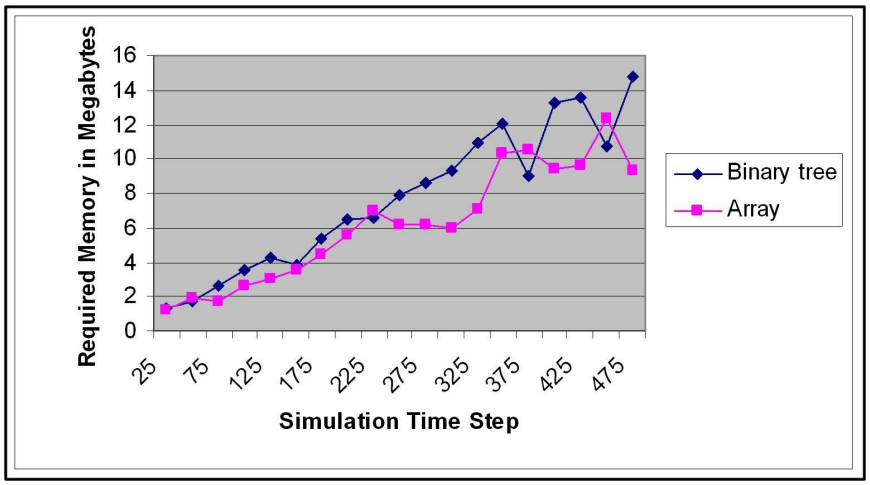

Figure 4.16: Comparison of existing strategy with proposed strategy based on memory requirement

Figure 4.16 shows the amount of memory needed for the proxel simulation at each time step for the implementations of the old and new approach. Most of the time, the array implementation has lower memory requirements than the binary tree implementation. Apart from removing the each proxel after processing, we also save 32 bits for each irrelevant entry of a supplementary variable. In some cases the existing implementation has lower memory requirements than the proposed implementation. They occur when the unused memory is greater than the memory savings in the array based implementation. More detailed experiments are made with other Petri nets to observe the runtime and memory requirements. Figure 4.17 shows another Petri net. The tokens in the Petri nets are increased for each time step.

- All transitions are uniformly distributed in the interval 7.0 to 9.0
- All of them belongs to age memory policy
- t=50 dt=0.25

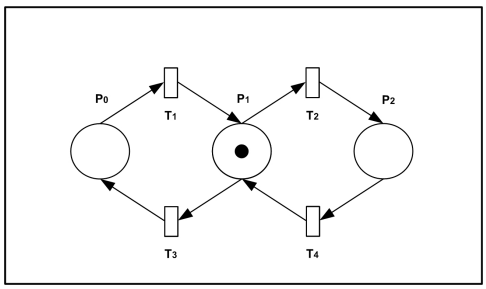

Figure 4.17: Example Petri net with three places

Figure 4.18 compares the computational time and the amount of memory needed for both storage strategies during proxel based simulation with one token in the Petri net. Therefore the number of markings of the Petri net is 3. There is no significant difference in the runtime but the memory of the array based implementation requires higher memory because of the proxel collisions and combinations.

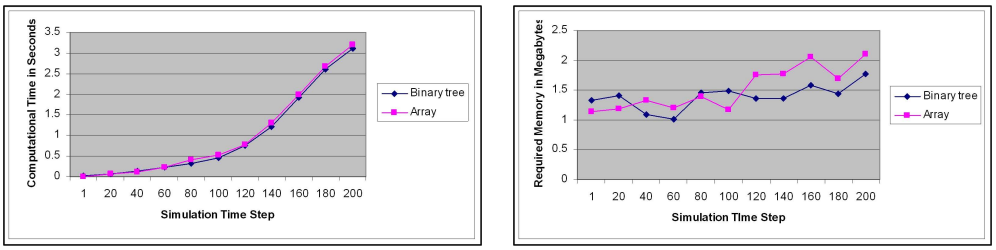

Figure 4.18: Comparison of storage strategies with one token in the Petri net

Figure 4.19 compares the runtime time and the memory requirement for both storage strategies during proxel based simulation with two tokens in the Petri net. Therefore the number of markings of the Petri net is 6. There is runtime of the array based implementation has shorter running time. The memory of the array based implementation requires higher memory in few time steps because of the same reason mentioned above.

The experiments are made with the same Petri net with 3 , 4 , 5, 6 tokens. Figures 4.20, 4.21, 4.22, 4.23 show the comparisons. In each of the case the number of markings increases but in all the cases the array based implementation has very shorter running time. The amount of memory needed for the array and binary tree implementations are also shown.

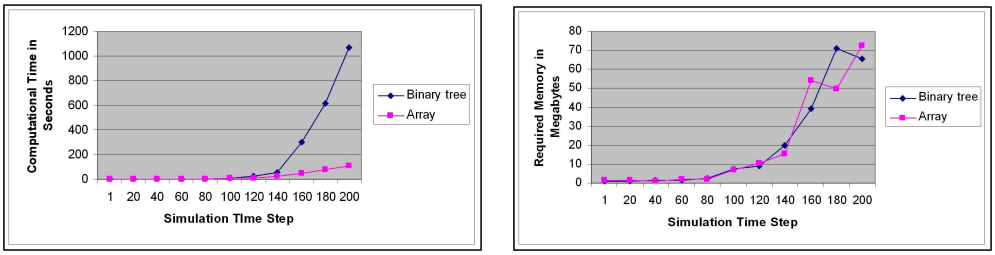

Figure 4.19: Comparison of storage strategies with two tokens in the Petri net

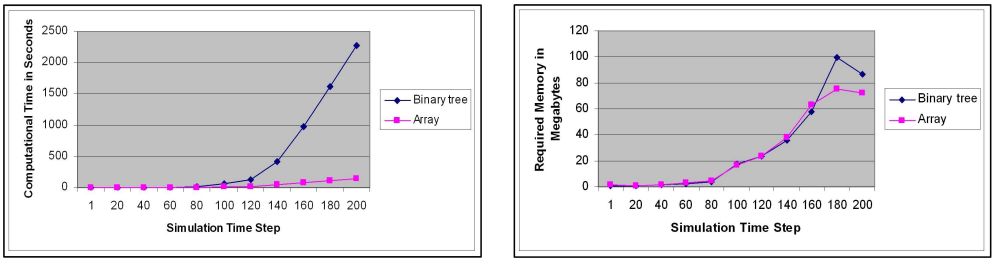

Figure 4.20: Comparison of storage strategies with three tokens in the Petri net

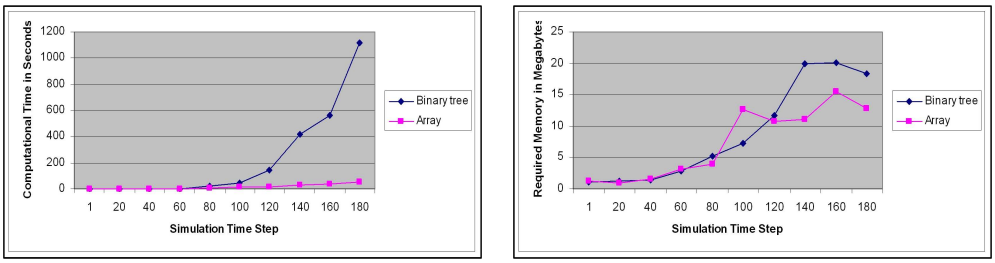

Figure 4.21: Comparison of storage strategies with four tokens in the Petri net

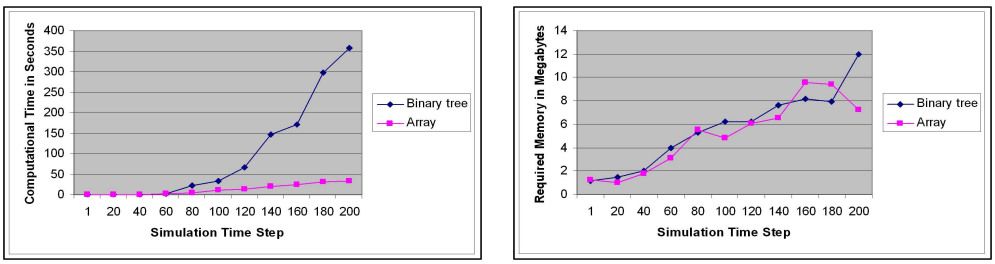

Figure 4.22: Comparison of storage strategies with five tokens in the Petri net

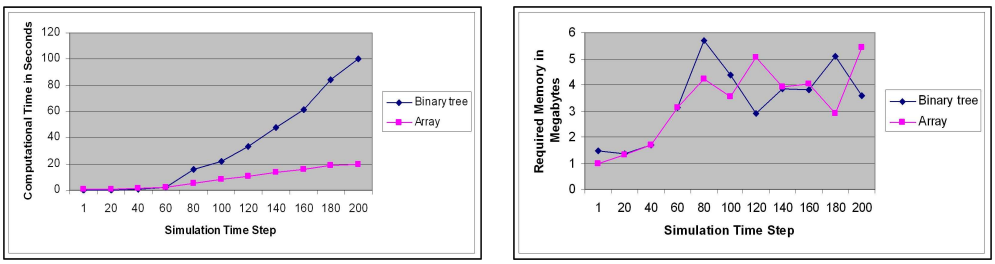

Figure 4.23: Comparison of storage strategies with six tokens in the Petri net

In all the experiments, the computational time of the array with hashing based implementation is shorter than the binary tree implementation. The memory requirements are slightly higher in few cases. From these experiments we conclude that, by having slightly higher memory requirements, we have achieved shorter runtime for this simulation approach.

Chapter 5

Conclusion and Future Work

This chapter describes the summary of the entire thesis with the contribution of the proposed approach. It is followed by the conclusion of the thesis and suggestions for improvement.

5.1 Summary

Preprocessing aims at automating the implementation of the proxel based simulation approach. This processes the Petri net and extracts the reachability graph for the proxel based simulation approach. The previous implementation was based on the reachability graph from the Petri nets. Therefore our first goal was to provide the current implementation with an automated generation of the reachability graph from the Petri net. The existing implementation is based on the constant size age intensity vector which needs a separate variable for each transition in the Petri net. This increases the key length and the memory requirements for each proxel. Therefore, the key computation time and memory requirements are higher during searching and storing a proxel. To overcome the above problem the proposed implementation defines marking dependent age intensity vector which defines the dimension of the age intensity vector.

The array implementation using a hashing technique has been suggested as an improved storage strategy for storage. We discussed the worst case and best case time complexity for array and binary tree implementations. The array structure supports the proxel search and inserts in a nearly constant access time. Therefore the new storage structure contributes to the current approach with shorter time for proxel search and insert operations. Further it avoids the recursive algorithms in the existing approach. This recursion is used for traversing binary trees in the current implementation.

5.2 Conclusion

A new approach to improve the design, storage and automation for the proxel based simulation has been developed. This approach generates the reachability graph from the Petri net and define the marking dependent age intensity vector required for proxel based simulation. Apart from automation, the implementation of this new approach reduce the required memory and computation time in the proxel based simulation. It saves memory by removing irrelevant supplementary variables from each marking. It saves time by making number of bits in the proxel's key smaller which is computed for storing the proxel in the data structure. Extensive experimental results have confirmed these results.

A new storage strategy for storing the proxels in the proxel based simulation has been designed. The data structure used has shorter running time for proxel based simulation than the existing implementation. However the memory utilization of the current implementation outperforms the new design in few cases but it is compromised with shorter running time of the new design. The improved space and time complexity of the new design are useful for the Petri nets containing number of discrete states. The experimental results have confirmed the theoretical results. They also have shown that the runtime of the proposed implementation is shorter than the existing implementation. Even though the memory requirement of the proposed implementation is larger in some cases, experimentations show only smaller differences. The proposed implementation achieves shorter runtime at the expense of slightly higher memory.

5.3 Future work

Every research work has a room for improvement. Even though the presented approach yields promising results, it can be improved to get better design and results than the proposed approach. We present some proposals in the following section, to improve the proposed approach.

Petri net specification: In the limitations section of the proposed design, we described the usage of common information interchange format. Since our motivation is to construct a general purpose proxel based simulator, it has to include a parser, to parse the Petri net specification from the Petri Net Markup Language. This removes our own specification of the Petri net [Jonathan Billington 2003].

Boundedness The current implementation of the proxel simulator works only for the bounded Petri net. Therefore it has to check and inform

the user that the Petri net is unbounded. The current implementation does not have this checking functionality. An algorithm to test the boundedness of the Petri net has to be included in the preprocessing step. Further the proxel based simulation does not have the boundedness limitation. Only the current and the proposed implementation lacks in supporting the unbounded Petri nets. Approaches which can handle unbounded Petri nets in proxel based simulation will improve the current and proposed approaches.

Proxel estimation The proxel estimation calculates the number of proxels for the next step. This takes only the number of enabled transitions. This allocate more memory for the proposed storage design. This estimation should include other factors such as proxel combination and threshold. If it is possible to determine the number of collisions in the next step, then that can be included in the proxel estimation process.

Proxel reusability We throw the proxels away from the memory after it is processed. But the same proxel can be generated after few steps. If we are able to keep track of the proxel and its successors then we can use the these proxels by recalculating the probabilities. Tracking all the proxels requires very high memory. Some tradeoff should be made in keeping the proxels in the memory for reusability. This can improve the running time of the simulation to a large extent but memory trade off needs more study.

Proxel based simulation is a very new approach for analyzing stochastic models. Even though the approach has a number of advantages, the implementations have limitations. This new approach needs further studies and research work for utilizing the maximum potential. With its full functionality, it might become one of the best approaches for certain applications in the simulation industry.

List of Figures

Bibliography

[German 2000] R. German, *Performance Analysis of Communication Systems. Modeling with Non-Markovian Stochastic Petri Nets, John Wiley and Sons, Ltd 2000.*

[Horton 2002] Horton, Graham, *A new paradigm for the numerical simulation of stochastic Petri nets with general firing times, Proceedings of the European Simulation Symposium 2002, Dresden. Society for Computer Simulation, 2002.*

[Lazarova-Molnar and Horton 2003 A] Lazarova-Molnar, S., and Horton, G.: *Proxel-Based Simulation of Stochastic Petri Nets. Submitted to the 10th International Workshop on Petri Nets and Performance Models, University of Illinois at Urbana- Champaign,2003.*

[Lazarova-Molnar and Horton 2003 B] Lazarova-Molnar, S., and Horton, G.: *An Experimental Study of the Behaviour of the Proxel-Based Simulation Algorithm. Simulation und Visualisierung 2003. Magdeburg: SCS Verlag: 2003.*

[Lazarova-Molnar and Horton 2003 C] Lazarova-Molnar, S., and Horton, G.: *Proxel-Based Simulation of Stochastic Petri Nets containing Immediate Transitions. On-Site Proceedings of the Satellite Workshop of ICALP 2003 in Eindhoven, Netherlands. Forschungsbericht Universität Dortmund. Dortmund 2003.*

[Gianfranco ciardo and Andrew S. Miner 2002] Gianfranco ciardo and Andrew S. Miner *Structural approaches for SPN analysis.*

[Jonathan Billington 2003] Jonathan Billington, Sren Christensen, Kees van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber *The Petri Net Markup Language: Concepts, Technology, and Tools.*

[Petri nets world 2004] Petri nets world *http://www.daimi.au.dk/PetriNets/.*

[Java 2 2004] Java 2 SDK, API specification *http://java.sun.com/j2se/1.4.2/docs/api/index.html.*

[Java Swings examples 2004] Using Swing components *http://java.sun.com/developer/onlineTraining/GUI/Swing1/.*

[Java Graph 2004] A powerful, lightweight, feature-rich, and thoroughly documented open-source graph component available for Java *http://www.jgraph.com/.*

[Java HotSpot VM Options 2004] Java Virtual Machine documentation and white papers*http://developers.sun.com/events/techdays/presentations/seattle /CodecampHotSpotVirtualMachineTuning.pdf.*

[Hash Functions for Hashtablelookup 2004] A Hash Function for Hashtablelookup *http://www.burtleburtle.net/bob/hash/doobs.html.*

[Java Memory allocations 2004] Java Technology Forums *http://forum.java.sun.com/thread.jsp?forum.*

[JDSL 2004] Java data structures library *http://www.cs.brown.edu/cgc/jdsl/.*

[64-bit Hash function] Java data structures library *http://www.serve.net/buz/hash.adt/java.000.html.*

[Algorithms and Data Structures] Dictionary of Algorithms and Data Structures *http://www.nist.gov/dads/.*

[Random distributions 2004] Resources for the professional in reliability engineering and related fields *http://www.weibull.com/*

[Ruskey 2004] Ruskey, F. "Information on Binary Trees." *http://www.theory.csc.uvic.ca/ cos/inf/tree/BinaryTrees.html*

[Skiena 1997] Skiena, S. S. The Algorithm Design Manual. New York: Springer-Verlag, pp. 177-178, 1997.

[Introduction to Hash funtions] Hash functions introduction from MathWorld *http://mathworld.wolfram.com/HashFunction.html*

[Balancing the binary trees] AVL tree and balancing *http://encyclopedia.thefreedictionary.com/AVLtree*

[Traversing binary trees] Methods of iterating over binary trees *http://encyclopedia.thefreedictionary.com/Fullbinarytree*

[David Eck and Bradley Kjel 2004] David Eck, and Bradley Kjel: Problems of recursion *http://micro5.mscc.huji.ac.il/ aries/Rec.html*